

1994

Causal synchrony in the design of distributed programs

Sandra L. Peterson

College of William & Mary - Arts & Sciences

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Peterson, Sandra L., "Causal synchrony in the design of distributed programs" (1994). *Dissertations, Theses, and Masters Projects*. Paper 1539623855.

<https://dx.doi.org/doi:10.21220/s2-mb5r-q596>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9511089

Causal synchrony in the design of distributed programs

Peterson, Sandra Louise, Ph.D.

The College of William and Mary, 1994

U·M·I

300 N. Zeeb Rd.
Ann Arbor, MI 48106

**CAUSAL SYNCHRONY
IN THE DESIGN OF DISTRIBUTED PROGRAMS**

A Dissertation

Presented to

**The Faculty of the Department of Computer Science
The College of William and Mary in Virginia**

In Partial Fulfillment

**Of the Requirements for the Degree of
Doctor of Philosophy**

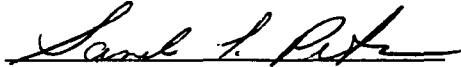
by

Sandra L. Peterson


1994

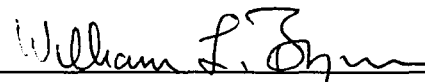
APPROVAL SHEET

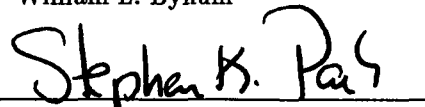
This dissertation is submitted in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy



Sandra L. Peterson


Approved, June 1994


John P. Kearns, Thesis Advisor


William L. Bynum


Stephen K. Park


Paul K. Stockmeyer


Roy L. Pearson

CAUSAL SYNCHRONY IN THE DESIGN OF DISTRIBUTED PROGRAMS

ABSTRACT

The outcome of any computation is determined by the order of the events in the computation and the state of the component variables of the computation at those events. The level of knowledge that can be obtained about event order and process state influences protocol design and operation. In a centralized system, the presence of a physical clock makes it easy to determine event order. It is a more difficult task in a distributed system because there is normally no global time. Hence, there is no common time reference to be used for ordering events. As a consequence, distributed protocols are often designed without explicit reference to event order. Instead they are based on some approximation of global state. Because global state is also difficult to identify in a distributed system, the resulting protocols are not as efficient or clear as they could be.

We subscribe to Lamport's proposition that the *relevant* temporal ordering of any two events is determined by their causal relationship and that knowledge of the causal order can be a powerful tool in protocol design. Mattern's vector time can be used to identify the causal order, thereby providing the common frame of reference needed to order events in a distributed computation. In this dissertation we present a consistent methodology for analysis and design of distributed protocols that is based on the causal order and vector time. Using it we can specify conditions which must be met for a protocol to be correct, we can define the axiomatic protocol specifications, and we can structure reasoning about the correctness of the specified protocol. Employing causality as a unifying concept clarifies protocol specifications and correctness arguments because it enables them to be defined purely in terms of local states and local events.

We have successfully applied this methodology to the problems of distributed termination de-

tection, distributed deadlock detection and resolution, and optimistic recovery. In all cases, the causally synchronous protocols we have presented are efficient and demonstrably correct.

SANDRA L. PETERSON

DEPARTMENT OF COMPUTER SCIENCE

THE COLLEGE OF WILLIAM AND MARY IN VIRGINIA

ADVISER: PHIL KEARNS

Contents

1	Introduction	2
1.1	Time and Order in Distributed Systems	2
1.2	Global Time and Clock Synchronization	4
1.3	Causality and Logical Clocks	7
1.3.1	Lamport's Logical Time	8
1.3.2	Vector Time	10
1.3.3	Causal Synchrony	14
2	Distributed Termination	17
2.1	Asynchronous Systems with Synchronous Communication	21
2.2	Fully Synchronous Systems	31
2.3	Asynchronous Systems and Asynchronous Communication	34
2.3.1	First-In, First-Out Channel Restrictions	34
2.3.2	Fully Asynchronous Systems	40
2.4	Causal Termination Detection Protocol	49
2.4.1	Vector Clocks for Synchronous Communication	49
2.4.2	Asynchronous System - Asynchronous Communication	56
3	Distributed Deadlock Detection and Resolution	64
3.1	Deadlock Detection - System Model	64

3.1.1	Previous Research	68
3.1.2	Deadlock and Causality	80
3.1.3	Synchronous Communication Protocols	84
3.1.4	Asynchronous FIFO Communication Protocols	92
3.1.5	Deadlock Resolution	93
4	Optimistic Recovery	103
4.1	Introduction	103
4.2	Previous Research	104
4.3	Rollback and Causality	114
4.3.1	Generic Model - Terminology	116
4.3.2	Historical Causality	117
4.3.3	Correctness Specifications and Polling Waves	125
4.4	Causal Recovery Protocol:Single Wave - Serial Failure	130
4.4.1	Informal Description	130
4.4.2	Formal Specification	133
4.4.3	An Example	137
4.4.4	Correctness	139
4.4.5	Commitment	147
4.4.6	Bounding the Size of <i>OrVect</i>	149
4.5	Real Time and Synchronous Recovery	150
4.5.1	Synchronous Rollback - Two Waves	154
4.5.2	Formal Specification	159
4.5.3	An Example - Synchronous Recovery	161
4.5.4	Correctness	163
4.6	Causal Recovery Protocol - Two Wave	167
4.6.1	Informal Description	168

4.6.2	Formal Specification	172
4.6.3	An Example	174
4.6.4	Correctness	177
4.7	Concurrent and Multiple Failures	183
4.7.1	Formal Specification	187
4.7.2	Correctness	189
5	Conclusions and Future Research	194

List of Figures

1.1	Concurrent Requests	6
1.2	Related Requests	7
1.3	Lamport's Logical Clocks	9
1.4	Vector Clocks	11
2.1	Control Wave Events	25
2.2	Construction of Φ set	36
2.3	Vector Time - Synchronous Communication	50
3.1	Data Manager and Transaction Execution	67
3.2	Transaction-Wait-For Graph	68
3.3	Initial Resource Allocation	70
3.4	Request for R_3	71
3.5	Condensed TWF Graphs	71
3.6	Counterexample: Initial Allocation	72
3.7	Counterexample: Second Phase	72
3.8	Counterexample: Deadlocked Set	73
3.9	Undetected Deadlock	78
3.10	False Deadlock	79
4.1	State Intervals	106

4.2	Example - Failure to Recognize Lost Messages	110
4.3	Example - Multiple Rollback	111
4.4	Failure and Rollback	116
4.5	Failed Execution History	119
4.6	Temporal and Causal History	120
4.7	Effect of Rollback on Causal Order	121
4.8	Persistence of Causal Effect of Message Transmission	123
4.9	Vector Time of Orphan Events	126
4.10	Causal Protocol - Single Wave	138
4.11	Causal Protocol - Single Wave	139
4.12	Commitment Protocol	148
4.13	Synchronous Protocol: Example	161
4.14	Resulting Consistent State	163
4.15	An Example	175
4.16	Resulting Consistent State	176
4.17	Concurrent Failures	185

ABSTRACT

The outcome of any computation is determined by the order of the events in the computation and the state of the component variables of the computation at those events. The level of knowledge that can be obtained about event order and process state influences protocol design and operation. In a centralized system, the presence of a physical clock makes it easy to determine event order. It is a more difficult task in a distributed system because there is normally no global time. Hence, there is no common time reference to be used for ordering events. As a consequence, distributed protocols are often designed without explicit reference to event order. Instead they are based on some approximation of global state. Because global state is also difficult to identify in a distributed system, the resulting protocols are not as efficient or clear as they could be.

We subscribe to Lamport's proposition that the *relevant* temporal ordering of any two events is determined by their causal relationship and that knowledge of the causal order can be a powerful tool in protocol design. Mattern's vector time can be used to identify the causal order, thereby providing the common frame of reference needed to order events in a distributed computation. In this dissertation we present a consistent methodology for analysis and design of distributed protocols that is based on the causal order and vector time. Using it we can specify conditions which must be met for a protocol to be correct, we can define the axiomatic protocol specifications, and we can structure reasoning about the correctness of the specified protocol. Employing causality as a unifying concept clarifies protocol specifications and correctness arguments because it enables them to be defined purely in terms of local states and local events.

We have successfully applied this methodology to the problems of distributed termination detection, distributed deadlock detection and resolution, and optimistic recovery. In all cases, the causally synchronous protocols we have presented are efficient and demonstrably correct.

CAUSAL SYNCHRONY IN THE DESIGN OF DISTRIBUTED PROGRAMS

Chapter 1

Introduction

1.1 Time and Order in Distributed Systems

A conventional single processor computer system is characterized by a single address space and by the presence of centralized control over the processes using the system. In addition, the occurrence of every atomic event in such a system can be ordered relative to each other on a common time line. Each of these characteristics - common memory, centralized control and the inherent ordering of events - influences the design of protocols and algorithms in a single processor system.

A distributed system lacks these characteristics. It is composed of independent processors operating concurrently, without central control. There is no common memory, and processes communicate only via message passing. While each process may have its own local clock, there is no global clock to be used as a common time reference for ordering the events which occur in the system.

It is the lack of a common time reference and its impact on protocol design that concerns us in this dissertation. Temporal ordering is essential to our way of thinking about computational problems and algorithms for their solution. Assumptions about the relative order of events are often integral to, and unstated in, protocol specifications and definitions.

This is the case because time is a basic component of our frame of reference, so basic that its availability is taken for granted.

A typical example of the prevalence of temporal ordering is the frequent use of first come first served (FCFS) scheduling algorithms in system design. Serving requests in the order in which they are made is a simple and fair technique, but it is based on a presumption that requests can, in fact, be ordered in time. The assumption that events can be ordered is so basic that it is not even stated as part of the specification.

In a centralized system the temporal ordering comes for free. Because there is no global clock in a distributed system, the temporal ordering of events is not available automatically. The inability to order events is a complicating factor requiring significant modification of centralized protocols so that they will work in a distributed environment. Mutual exclusion [1, 2, 3, 4] and deadlock detection [5, 6, 7, 8, 9, 10, 11, 12] are examples of problems for which there are straightforward centralized solutions which will not work properly in a distributed environment. Problems such as termination detection [13, 14, 15, 16, 17, 18, 19, 20], election [21, 22, 23, 24, 25], and agreement [26, 27, 28, 29], which are specific to distributed systems, also arise, in part, because of the absence of global time. Lack of a central clock is not the only aspect of distributed systems that is a source of difficulty. The independent nature of the processes and lack of common memory play a part as well. However, the ability to establish some sort of temporal ordering of events is a necessary part of the solution to any of these problems.

Mutual exclusion is an example of a problem which is relatively easy to solve in a centralized system. The mutual exclusion paradigm requires that a process, when granted mutual exclusion, has exclusive access to some resource. Normally this exclusive right is granted in the same order that requests for the resource are made, i.e., in FCFS order. In a centralized system the operating system grants the requests, and a simple queue guarantees the appropriate ordering.

In a distributed system the solution is complicated by two factors. One, there may be

no controlling process to grant mutual exclusion requests, and two, without global time, it is difficult to identify which request is “first”. Creating a semblance of central control by endowing one process with the power to enforce mutual exclusion is not in itself adequate to solve this problem. Some way of establishing the temporal order of requests is also required.

The lack of global time necessitates that distributed protocols indirectly establish the temporal ordering needed to solve a problem like FCFS mutual exclusion. The indirect approach usually complicates the protocols by increasing the amount of communication between processes. Protocols for distributed mutual exclusion provide a clear example of this. Those protocols which provide first-come, first-served mutual exclusion [1, 2] generally require $O(N)$ messages per service, where N is the number of processes in the system. Protocols which grant mutual exclusion in an unspecified order [3, 4] require as little as $O(\sqrt{N})$ messages. The extra messages are a direct result of needing to broadcast a request to all processes to establish order between requests. Availability of global time would simplify distributed protocols by allowing protocols to be designed as if temporal ordering were directly given in a manner analogous to centralized systems, thus eliminating the messages needed to establish it indirectly.

1.2 Global Time and Clock Synchronization

The beneficial aspect of having a global clock in a distributed system has been acknowledged for some time. To this end, numerous protocols ([30, 31], for example) have been proposed for synchronizing clocks and providing an approximate global time. These protocols generally rely on periodic communication between all the processes to determine the amount of drift which has occurred in each process' clock. Based on these determinations, a correction factor is calculated for each clock to counteract the drift and to keep each clock synchronized with all the clocks in the system.

The difficulties with this approach are three-fold. First, in most clock synchronization

protocols each execution of the algorithm requires $O(N^2)$ messages, where N is the number of processes participating in the protocol. Second, most of these protocols require that message transmission time be bounded by some identifiable constant. While this may be a reasonable requirement in a system attached to a local area network, it is not a general solution applicable to geographically dispersed processors. Finally, none of these algorithms provide absolute accuracy. This may not matter for some protocols; however, any protocol which uses one of these synchronization algorithms as a method for establishing global time must be designed with these inaccuracies in mind.

Clock synchronization seems best suited to a small group of processors connected to a local area network and to applications which are not affected by small errors in the global time approximation. As a *general* solution for distributed systems these protocols are unrealistic and computationally expensive.

An appropriate use of synchronized clocks is to enable processors to act in concert at a specified time or after a predetermined interval. Synchronized clocks can also be used to identify the ordering of events. For example, physical clocks can be used to solve the ordering problem in a distributed protocol for mutual exclusion. Each process could timestamp its request, and these timestamps could be used to order the granting of the requests.

The presence of timestamps does not totally solve the distributed mutual exclusion problem. There is no entity with a global view of the system which can observe all the request times simultaneously for the purpose of granting requests in the appropriate order. This part of the problem can be solved in a variety of ways. One process can act as a central server to which all requests are sent. Because of message transmission delay, the server would either have to communicate with processes to assure that they had no requests in transit, or it could age the request some time period large enough to assure that any earlier request had time to arrive. A distributed algorithm which uses physical timestamps would require that each process broadcast its timestamped request to all the other processes. The appropriate ordering would then be determined locally at each process.

A mutual exclusion protocol based on physical timestamps would guarantee that if $T(r)$ is the physical time of request r , and if r_1 and r_2 are two requests for which $T(r_1) < T(r_2)$, then r_1 would be honored before r_2 . This is a logical requirement in a centralized system. In a distributed system it may be a more stringent requirement than necessary. In a distributed system it is possible for $T(r_1) < T(r_2)$, yet it is not possible for r_1 to affect r_2 . In the time and space diagram [2] shown in Figure 1.1, there are two processes, p_0 and p_1 . Request r_1 occurs in p_0 , and r_2 occurs in p_1 . There is no communication between p_0 and p_1 , so r_1 can have no impact on r_2 . While they are ordered in physical time, r_1 and r_2 are considered “causally” concurrent, and their temporal ordering is irrelevant from a computational point of view. The requests could be granted in either order and the outcome would be correct.

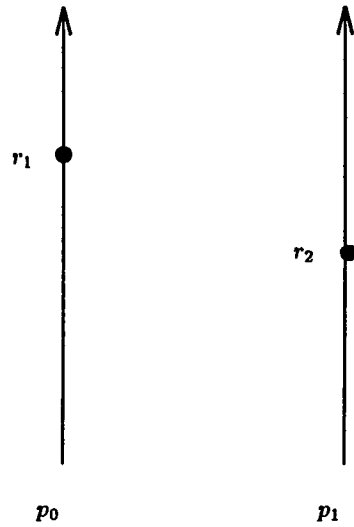


Figure 1.1: Concurrent Requests

Now consider Figure 1.2. In this example p_0 makes request r_1 and follows it with a message to p_1 . p_1 then proceeds to make its own request, r_2 . In this case it is possible for the events that occurred in p_0 up until the transmission of the message to have an effect on p_1 . Therefore, it makes sense to say that r_1 *causally precedes* r_2 , and the requests should

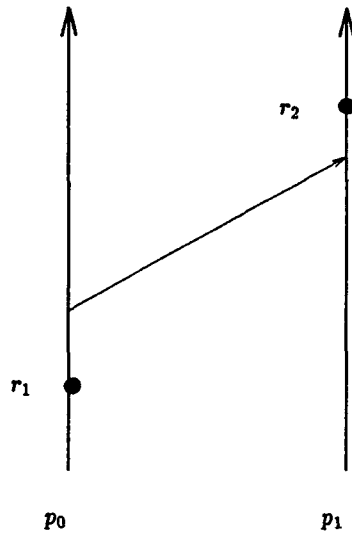


Figure 1.2: Related Requests

be granted in that order.

1.3 Causality and Logical Clocks

In a distributed system there is a logical ordering of events induced by causality. The relevant order of events is really determined by whether one event can potentially affect another one. Physical time does not fully convey this aspect of distributed event temporal ordering. The timestamps of two events can be used to show that one event can't affect another; hence if $T(e_1) < T(e_2)$, e_2 can not affect e_1 . Physical timestamps can't be used to show that one event can affect another; $T(e_1)$ may be less than $T(e_2)$, but in fact the events may have no causal relationship to each other. For the problem of mutual exclusion there is no need to distinguish between concurrent events and ones that are causally related. It is sufficient to require that the smallest timestamp is served first.

1.3.1 Lamport's Logical Time

An alternative to physical, synchronized clocks is a system of *logical* clocks such as those developed by Lamport [2]. He formalized the notion of causality to which we alluded above. Lamport recognized the value of having some consistent definition of time in a distributed system, so that it would make sense to talk about the relative order of events. He also realized that, while global time is not available in asynchronous systems, an identifiable temporal ordering does exist between two events in the same process and between events in processes which communicate; this ordering is based on potential causality. Using causality as a basis, it becomes unnecessary to totally order all the events which occur. Some process events happen concurrently and can have no effect upon one another. The only events which need to be ordered are those which can influence each other. Based on this observation Lamport defined the “happens before” relation \rightarrow which formally defines the causal ordering relevant in a distributed system.

This relation is defined as the smallest relation such that:

1. If event e_1 and e_2 are in the same process, and e_1 occurs before e_2 , then $e_1 \rightarrow e_2$;
2. If an event e_1 is the sending of a message in process p_1 , and e_2 is the receipt of this message in process p_2 , then $e_1 \rightarrow e_2$;
3. If $e_1 \rightarrow e_2$, and $e_2 \rightarrow e_3$, then $e_1 \rightarrow e_3$.

A pair of events is concurrent, signified by $e_1 \parallel e_2$, if they are not ordered under \rightarrow . Therefore, $e_1 \parallel e_2$ if and only if $(e_1 \not\rightarrow e_2) \wedge (e_2 \not\rightarrow e_1)$. This is consistent with the notion that concurrent events are those events which can have no effect on each other.

Lamport defines a logical clock function, C , based on a counter maintained by each process. The function assigns times to events that are consistent with this causal ordering. If $C(e_1)$ and $C(e_2)$ are the logical times assigned to events e_1 and e_2 , and $e_1 \rightarrow e_2$, then $C(e_1) < C(e_2)$. The rules for maintaining such clocks are straightforward. Each process,

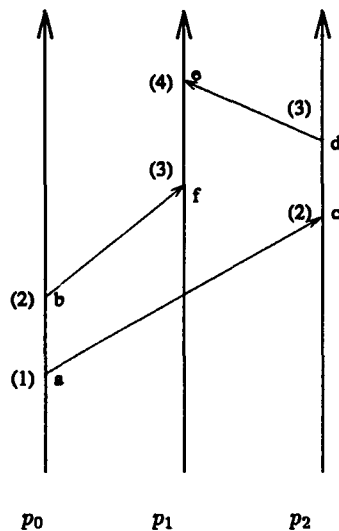


Figure 1.3: Lamport's Logical Clocks

p_i , maintains a counter C_i . C_i is incremented each time an event occurs in p_i . The logical time associated with an event e is denoted by $C_i(e)$ and equals the value of p_i 's counter at the time e takes place. If e_1 is a send in p_i , and e_2 is the corresponding receive in p_j , $C_i(e_1)$ is sent along with the message. $C_j(e_2)$ is set so that it is greater than $C_i(e_1)$ and greater than or equal to the value of C_j when e_2 occurred.

An example of the operation of Lamport's logical clocks is shown in Figure 1.3. The letters indicate the various send and receive events. The numbers in parentheses are the associated clock values. In this example $a \rightarrow b$, and $C(a) < C(b)$. Note also that $a \rightarrow e$ (by the transitivity of \rightarrow); therefore, $C(a) < C(e)$.

Lamport uses his logical clocks to design a protocol for FCFS mutual exclusion. We will restate it briefly here to illustrate how logical clocks can be used as a substitute for physical time. In Lamport's algorithm each process maintains a queue of timestamped requests that it has received, including any request that it has made itself. When a process makes a request, it broadcasts a timestamped request message to every other process. Any

process receiving a request sends a timestamped acknowledgment to the requesting process. A process is granted mutual exclusion when its request is the lowest timestamp in its queue, and it has received a higher timestamped message from every other process. When a process releases the resource, it notifies every process so that the satisfied request can be removed from the process queues.

This protocol guarantees that if $r_1 \rightarrow r_2$, then r_1 will be granted first. A protocol based on synchronized clocks could be easily developed by substituting physical time for logical time in Lamport's algorithm. In the case of this protocol, logical time and physical time provide the same capability for ordering requests.

Lamport's clocks and physical clocks have a similar deficiency; the clock values can only be used to determine that two events could not have occurred in a certain order; $C(e_1) < C(e_2)$ implies that e_2 could not have happened before e_1 . Based on the clock values it is not possible to state the converse, that e_1 happened before e_2 . Lamport clock values can not be used to distinguish between $e_1 \rightarrow e_2$ and $e_1 \parallel e_2$. As we pointed out when talking about physical clocks, not being able to make this determination does not matter for the mutual exclusion problem, but it is a deficiency that the clocks do not totally correspond to the partial order.

1.3.2 Vector Time

Mattern [32] and, concurrently, Fidge [33] implement logical clocks using vectors of counters with the object of creating clock values which capture the complete partial order of events in a distributed system. The extra information preserved in the vector clocks allows the ordering of events to be deduced from the relative timestamp values in a way that Lamport's clocks do not.

Suppose that a distributed program consists of the set of processes $\{p_0, p_1, \dots, p_{N-1}\}$. Each process, p_i , has a vector clock $V_i^j, 0 \leq j \leq N - 1$. $V_i(e)$ indicates the clock value of an event e which has occurred in p_i . The i^{th} component of the vector is incremented before

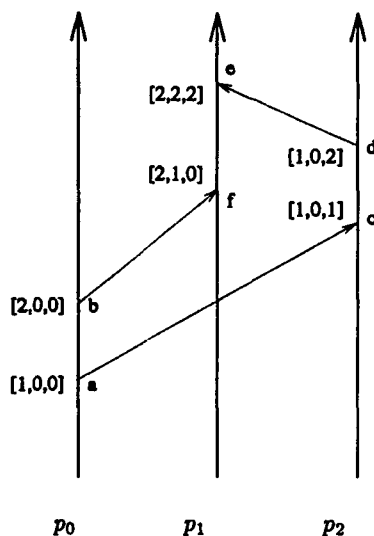


Figure 1.4: Vector Clocks

each event in process p_i , and the current timestamp vector is sent along with each message to update the receiving process' clock. More formally, the rules for maintaining the vector clocks are as follows:

1. When event e occurs in p_i , $V_i^i = V_i^i + 1$. The clock value of e is $V_i(e)$.
2. If e_1 is a send in p_i , and e_2 a receive in p_j , then the clock value of e_2 is updated to reflect the clock value of e_1 so that $V_j(e_2)$ is assigned $\text{sup}(V_j, V_i)$, where $\text{sup}(V_i, V_j) = \max(V_i^k, V_j^k)$, for $0 \leq k \leq N - 1$.

Figure 1.4 shows how the vector time would be calculated for the sample computation shown in Figure 1.3. The value of V_i^i is always equal to the number of events that have occurred in p_i . The value of V_i^j , where $j \neq i$, indicates the most current information that p_i has about p_j 's activity. This information may be outdated. However, from p_i 's point of view as obtained from messages sent to p_i , it is the most up-to-date and causally relevant information available.

Mattern defines the following relationships between vectors for any two clock values V_i and V_j .

1. $V_i \leq V_j$ iff $\forall k : V_i^k \leq V_j^k$
2. $V_i < V_j$ iff $V_i \leq V_j$ and $V_i \neq V_j$
3. $V_i \parallel V_j$ iff $(V_i \not\leq V_j) \wedge (V_j \not\leq V_i)$

Using these relationships, Mattern shows how vector timestamps can be used to determine the causal order between any pair of events e_1 and e_2 :

1. $e_1 \rightarrow e_2$ iff $V_1(e_1) < V_2(e_2)$
2. $e_1 \parallel e_2$ iff $V_1(e_1) \parallel V_2(e_2)$

By allowing us to distinguish between $e_1 \rightarrow e_2$ and $e_1 \parallel e_2$, vector clocks supply extra information about the system. The question is whether this information is useful in developing distributed solutions to computational problems.

An example of how vector time can be exploited in protocol design is the vector time variant of the algorithm for providing first-come first-served centralized service proposed by Kearns and Koodalattapuram [34]. The protocol they present has a centralized server grant exclusive resource access to requesting processes, in the causal order in which the requests are made. This is a variation of FCFS mutual exclusion as it applies to managing a centralized resource. Their protocol differs from the centralized mutual exclusion algorithm we briefly described earlier in that it can determine when to serve a request based solely on the timestamp of the request and information stored locally in the server. It is not necessary either to age the requests or to communicate with any processes to determine whether they have earlier outstanding requests.

In this algorithm vector time is updated on each request for service. When p_i makes a request, V_i^i is set to $V_i^i + 1$. This vector timestamp is appended to the requesting message

sent to the server. Any process which communicates with another process also appends its timestamp to the message it sends. A process, p_i , which receives a message from p_j updates its own timestamp so that $V_i = \text{sup}(V_i, V_j)$. The server also maintains a vector V_c . V_c^i is set to the value of V_i^i from the vector attached to the last request message from p_i to be processed at the server.

With this information the server can tell whether it has received a message out of order by comparing V_c to the vector attached to the incoming request. If $V_i^j > V_c^j$ for any $j \neq i$, then the request from p_i is early; it has reached the server before a service request which causally precedes it. The server knows to place these early messages “on hold” until the causally earlier requests arrive. If $V_i^j \leq V_c^j$ for all $j \neq i$, then the server can be sure that there are no outstanding requests which causally precede the request from p_i . This request can be served immediately.

The server can make this decision because the contents of V_i capture the state, as far as outstanding requests are concerned, of all the processes which communicated with p_i before p_i made its request. Since these outstanding requests are the only ones that can causally affect p_i , the server knows that p_i 's request can be safely granted once they have been served. Since the server has this information locally in V_c , it can immediately decide whether it should serve p_i 's request.

This algorithm is very interesting in that it illustrates an aspect of vector time which makes it more powerful than either physical time or Lamport's logical time. The protocol shows that vector timestamps transmit information about the process which creates the timestamp, and in this case, that information was enough to eliminate the need to communicate with processes about their state. Neither physical clocks, or Lamport's logical clocks provide this information.

1.3.3 Causal Synchrony

As the examples of FCFS service and mutual exclusion illustrate, the ability to determine event order is invaluable in the design of distributed protocols. Given that global state does not exist in a distributed system in the same sense it does in a centralized system, event order provides a common frame of reference on which to base protocol behavior.

In a synchronous system, the temporal order of events can be identified through the use of synchronized real time clocks. Knowledge of the temporal order has been shown to be a powerful tool for simplifying protocol design [35, 36], however, synchronized clocks are difficult and expensive to implement. In an asynchronous system, the temporal order can not be determined. However, the causal order of events can be identified and utilized through the mechanism of vector time. In this way, vector clocks can provide an illusion of synchrony in an asynchronous system—a kind of pseudo-synchrony which is based on causality. For many distributed protocols we believe this pseudo-synchrony, or *causal synchrony* as we prefer to call it, is an adequate substitute for true synchrony.

Vector clocks and causal order are not an appropriate substitute for synchronized clocks and temporal order in every case. For example, a protocol which requires the performance of some action at a specific real time can not directly substitute a vector clock time for a real clock time. Some distributed problems only require the determination of relative event order for their solution. In these cases the similarities between the causal order and the temporal order are great enough that vector clocks can be a practical substitute for real clocks.

The algorithm for FCFS centralized service, discussed earlier, shows how knowledge of the causal order is sufficient to structure an effective protocol. Other protocols using vector clocks and causal order have appeared in the literature. Mattern proposes using them to obtain global snapshots. Both Fidge and Mattern advocate their use in distributed debugging. Ahamad, et.al [37] used vector clocks to implement a weakly consistent distributed

shared memory.

In this dissertation we develop causally synchronous distributed protocols for termination detection, deadlock detection and resolution, and optimistic recovery. In each case we begin with a solution which presumes that synchronized clocks are available. As expected, the resulting protocol is relatively simple and efficient. Using the synchronous protocol as a model, we design a corresponding causally synchronous protocol that uses vector clocks instead of synchronized clocks. While substitution is not exact, the structure of the synchronous and causally synchronous protocols is very similar. More importantly, using the synchronous model as a pattern consistently leads to more efficient protocols for these problems than previously published solutions.

In Chapter 2 we present several causally synchronous solutions to distributed termination detection. This is a well known problem with several published solutions. Most of these solutions rely on indirect methods for establishing the necessary temporal ordering. However, a protocol proposed by Rana [16] uses synchronized clocks. This presented us with the opportunity to show how a protocol which utilizes global time could be readily implemented using vector time as a substitute for real time. We found that reasoning about the existing termination detection protocols in a causal way was extremely productive, both for developing a consistent taxonomy for the extant solutions and for developing our own vector time protocols.

In Chapter 3 we discuss the problem of resource deadlock detection and resolution, and we present a causally synchronous protocol for its solution. Detecting and resolving deadlock in a distributed system is a difficult problem that has been extensively researched with unsatisfactory results. The use of vector time enabled us to design a straightforward and demonstrably correct protocol to detect and resolve distributed deadlock.

Chapter 4 is devoted to optimistic recovery. Optimistic recovery is a technique for providing fault tolerance to a distributed system. The problem of recovery is a difficult one because of the impact of failure and the resulting loss of state on the causal order and

the consistency of vector clocks. Once again, considering the problem in a synchronous environment provided a suitable model into which vector clocks could be substituted to provide a solution in an asynchronous environment.

In the final chapter, Chapter 5, we present our conclusions, summarize our findings, and discuss future areas of research.

Chapter 2

Distributed Termination

One of the problems unique to distributed systems is determining that a distributed computation has terminated. The concept of termination of a sequential program on a single processor is well defined. Detecting termination in this case can be accomplished based solely on the program's state information. What it means for a distributed program to be terminated is not as easily defined. One approach is to model a distributed program as many sequential programs operating on autonomous processors. Based on this model, a termination requirement which comes immediately to mind is that the local state of each process in the program be in accordance with some completion criteria at some time instant. However, in a distributed system there is normally no availability of global time, so such a definition is nonsensical. Further complicating matters is that in some computations a local process can not tell, based on its local state, whether it is terminated, or just temporarily idle. Whether or not a process is permanently idle may depend on the state of another process or set of processes. As a result of this interdependence it is not enough to define termination based on the state information of an individual process in the system. The state of every process must be evaluated to determine if termination has occurred. For this reason termination is viewed as a characteristic of the *global* state of the system and is usually defined in terms of the global state.

Francez [13] was one of the first to formally describe the distributed termination problem. The global state of a distributed computation is defined in terms of a global predicate B^* , such that a computation is terminated when B^* is satisfied. He presumes that the computation can be structured so that for each process, p_i , there is defined a local predicate, b_i , such that b_i true for all i implies B^* .

Testing the local predicates can then be used to determine the global state of termination. The method used for testing must be designed carefully. Simply checking each process to see if the local predicate holds will not work because the truth of a local predicate may be altered by the actions of another process. The fact that process p_j has satisfied its local predicate b_j does not necessarily mean that p_j is terminated. It is possible for other processes which are still active to negate b_j by sending p_j a message. For this reason it is necessary to test the local predicates in such a way as to assure that they are all satisfied *simultaneously*. This is a significant weakness in the definition. Without a concept of global time this is a theoretical requirement that cannot be verified in a real system.

Dijkstra, et al. [15] also attack the problem of distributed termination. Rather than defining this problem in terms of predicates on the local process states, they make no assumptions about an individual process' ability to tell whether it has met a specific termination requirement. Instead their definition only requires that a process determine whether it is active or idle. An idle process cannot become active spontaneously. Once idle it remains so until it receives a message from an active process. Only active processes can send messages. Given these conditions, termination in a distributed system is said to have occurred when all processes are idle, and there are no messages in transit. Note there is also an implied time frame in this definition. Not only must all processes be idle, the implication is that they must all be idle at the same time.

Because it is more general, Dijkstra's characterization of the termination detection problem subsumed the concept of local predicate to become the standard paradigm used in development of subsequent termination detection algorithms.

Both of these definitions of the termination of distributed computation imply that knowledge of the global state of the system is needed to detect termination. No single process has access to this information, so it becomes necessary to superimpose a detecting computation onto the basic program which will insure that some process will gain the requisite knowledge about global state. It is relatively easy to gather information about the local state of each process. The difficult part of such a detecting computation is to verify that every process is idle at the same time when an accurate global clock is not available. To be correct, such a detection algorithm should conclude a program is terminated if and only if the underlying computation is terminated. The superimposed algorithm should interfere as little as possible with the underlying computation, and it should conclude termination within a reasonable time period after termination has occurred.

Appropriate solutions to the termination detection problem depend upon the specifics of the system environment. The distributed environment is usually asynchronous. In an *asynchronous system*, no effort is made to synchronize local process clocks to other clocks in the system. As a result, global time is not defined. A termination detection protocol meant for an asynchronous system must accommodate this lack of global time. An alternative system environment is one which is *synchronous*. In such a system, a global notion of time is available to each process. This is accomplished by imposing some type of clock synchronization protocol on the system. The complexity of detecting termination is reduced somewhat in a system where such synchronized clocks are available.

The semantics of message passing also impact the solution technique which is appropriate for detecting termination. A common system specification for existing termination detection protocols assumes asynchronous operation of the processors, but it requires that all communication is synchronous. Such a system is termed *asynchronous with synchronous communication*. Hoare [38] developed this communication paradigm in which a communicating process blocks until the corresponding sending or receiving process is ready to communicate. Synchronous communication is usually implemented by having a sending

process block until it receives acknowledgment from the receiving process. As a result, a process cannot proceed until it knows that the message has been received. A receiving process must also block until it receives a message. Imposing the restriction of synchronous communication on the system also reduces the complexity of the termination detection algorithm.

It is also possible to specify a system environment that is synchronous and supports synchronous communication. This system type will be referred to as *fully synchronous*. A synchronous system in which communication is not synchronous will be termed *synchronous system with asynchronous communications*. Neither of these system types are commonly used as a basis for design of termination detection algorithms. This is probably the case because of the expensive and unrealistic requirement of synchronized clocks.

Another system type which is used in outlining termination detection algorithms is a collection of asynchronous process which also communicate asynchronously. No process has access to global time, and after sending a message, a process continues with its activity without waiting for an acknowledgment. An important restriction is generally placed on communication between any two processes. This restriction is that messages are transmitted in a well ordered manner so that any two messages sent on a single channel between a set of two processes must be received in the same order in which they are sent. Such a system is termed *asynchronous with First-In First-Out channels*.

The most general system environment is one in which the processes are asynchronous and the only restriction placed on message transmission is that any message sent is received a finite length of time. We will designate such a system as *asynchronous with asynchronous communication*.

In the following sections, we will organize our discussion of existing algorithms according to the type of system for which they are applicable. In each of these algorithms, messages which result from the detecting computation are considered separate from those in the underlying computation. An idle process which receives or sends a detecting computation

message is not considered active. Only basic communication, messages resulting from the underlying computation, can activate a process.

2.1 Asynchronous Systems with Synchronous Communication

In this environment the standard paradigm is to repeatedly poll the processes in the system until every process reports that it has been continuously idle since the last poll. We will discuss two of the existing algorithms in detail to illustrate how this paradigm is applied.

Francez and Rodeh [14] propose an algorithm which uses a spanning tree as the organizational structure of communication for the detecting computation. The spanning tree is constructed from existing communication links in the system. The root of this tree is the process which detects termination. The leaves of the tree are responsible for initiating the detecting computation.

When a leaf becomes idle, it propagates a control message to its parent. The parent, upon receiving a control message from each leaf, and upon becoming idle itself, propagates the control message to its parent. This process is repeated for each subtree. A subtree root propagates the control message to parent when it has collected control messages from all of its children and is itself idle. In this way a *control wave* spreads up through the tree as nodes become idle, until it reaches the root of the spanning tree.

The control wave reaching the root is not sufficient to declare termination. Every process is idle when polled, but the processes are not polled simultaneously. It is possible for a process which has not yet been polled to send an activating message to a process which has already been passed by the control wave. This kind of activity must be detected or termination will be falsely declared. To detect this activity each process sets a flag to true if it participates in any communication unrelated to the detection algorithm. The control messages propagated by the processes also have a true or false value. When any process

propagates the control wave it sets the value of the control message to that of its flag and resets its flag to false. If a node has been active since the last wave the status of the flag will be true. By including this status in the subsequent wave message, this information will be carried to the root. A wave which collects at least one true flag is considered unsuccessful. An unsuccessful wave causes the root to echo a restart control wave outward to the leaves. The detecting wave is restarted by the leaves when they receive the restart control wave.

A control wave which reaches the root without collecting any of these true flags is considered a successful wave. Receipt of a successful wave at the root implies that every process has been continuously idle since the last wave. In this environment every process being continuously idle between two waves implies termination. This is the case because all communication in this algorithm is synchronous. This guarantees that any activating message sent during one wave will be received before the next wave; i.e., a message cannot cross two waves. We will show later that this is absolutely necessary for the correct operation of the protocol.

Dijkstra[15] derives a similar solution using a circulating token to generate the detection control wave. In this algorithm the polling structure imposed on the processes is a virtual ring, with a specified node acting as both the initiator of the detection wave and as the process which determines whether termination has occurred. In a system of N logical processes, the processes are numbered from p_0 to p_{N-1} . A token circulates in the ring such that p_0 sends the token to p_{N-1} and p_{i+1} sends the token to p_i for $0 \leq i \leq N-2$. When the initiating node, p_0 , becomes idle it generates a token and sends it to the next process in the virtual ring. Any process receiving the token propagates it in turn when it enters an idle state. The control wave is considered complete when the token returns to p_0 .

As in the Francez and Rodeh algorithm, it is not sufficient that the token returns to the initiator having left each process in an idle state. It is also necessary that all processes remained idle after they had been polled. Because the processes are not polled simultaneously, it is possible for processes ahead of the control wave to generate activity behind the

control wave thus violating this requirement.

To detect this activity a process engaging in basic communication is flagged by coloring itself black. The token, when propagated by a black process, is also colored black. The propagation of the token colors the process white as it passes. A black token returning to p_0 indicates that the control wave has failed, and a new wave is started by coloring the token white and sending it to p_{N-1} . A wave is successful when a white token returns to p_0 .

There are striking similarities between the algorithms proposed in [14] and [15]. A process coloring itself black in [15] corresponds to a process setting its flag to true in [14]. Coloring the token black when it passes a black process serves the same purpose as changing the status of the control message to true. Finally, resetting the process color to white with the passage of the token corresponds to resetting the process flag to false when the control message is propagated. The primary difference in the two techniques is the method used to poll the processes.

At least two other algorithms have been proposed to detect termination in an asynchronous system which uses synchronous (or "instantaneous") message transmission, [17, 18]. The operation of these protocols is similar to those we have already discussed. In fact, the basic concept behind all of these algorithms is the same. Every process is polled to determine whether it is idle. Because it is impossible to simultaneously poll every process, it is possible for a process which is yet to be polled to activate a process which has already been polled. This is detected by marking the sender of a message so that it will be known that there was a message in transit during the poll. The recipient is also marked so that its activation will be noticed. Any activity detected by a poll will result in another polling wave. Termination is detected when a wave completes showing no communication activity since the last wave.

Each of these algorithms is dependent on synchronous message passing to work correctly. The synchronous nature of the communication guarantees that any activating messages sent before a wave arrives will be received, and noticed, before the next wave arrives. Without

this guarantee these algorithms would be incorrect.

To more formally describe this general method in causal terms, we need to define some terminology and use it to restate the definition of termination so that it is applicable to this protocol type.

- $\Pi = \{p_0, p_1, \dots, p_{N-1}\}$ is the set of processes in the distributed computation.
- e'_i is a generic event in p_i .
- s represents a send event of the underlying computation.
- $\eta(s)$ represent a receive event corresponding to a transmission s .
- $\sigma(s)$ is the process where send event s occurs.
- $\rho(s)$ is the receiving process for a send event s .
- $c_j(i)$ signifies the event of the i^{th} control message arriving at p_j .
- $w_j(i)$ represents the process event which occurs when p_j responds to control message $c_j(i)$.
- $PW(i)$ is the i^{th} polling wave. $PW(i) = C(i) \cup W(i)$,
where $C(i) = \{c_0(i), c_1(i), \dots, c_{N-1}(i)\}$, and $W(i) = \{w_0(i), w_2(i) \dots w_{N-1}(i)\}$.
- $e'_i \mapsto e''_i$ iff there does not exist e'''_i such that $e'_i \rightarrow e'''_i \rightarrow e''_i$.
- $ie_j(k)$ signifies the event of p_j going idle for the k^{th} time.
- $Idle()$ is a function from events to $\{\text{True}, \text{False}\}$.

Relating this terminology to Dijkstra's protocol, the event $c_j(i)$ corresponds to the token arriving at p_j for the i^{th} time. The action of the token leaving p_j for the i^{th} time is signified by $w_j(i)$. The i^{th} complete circuit of the token corresponds to the polling wave $PW(i)$.

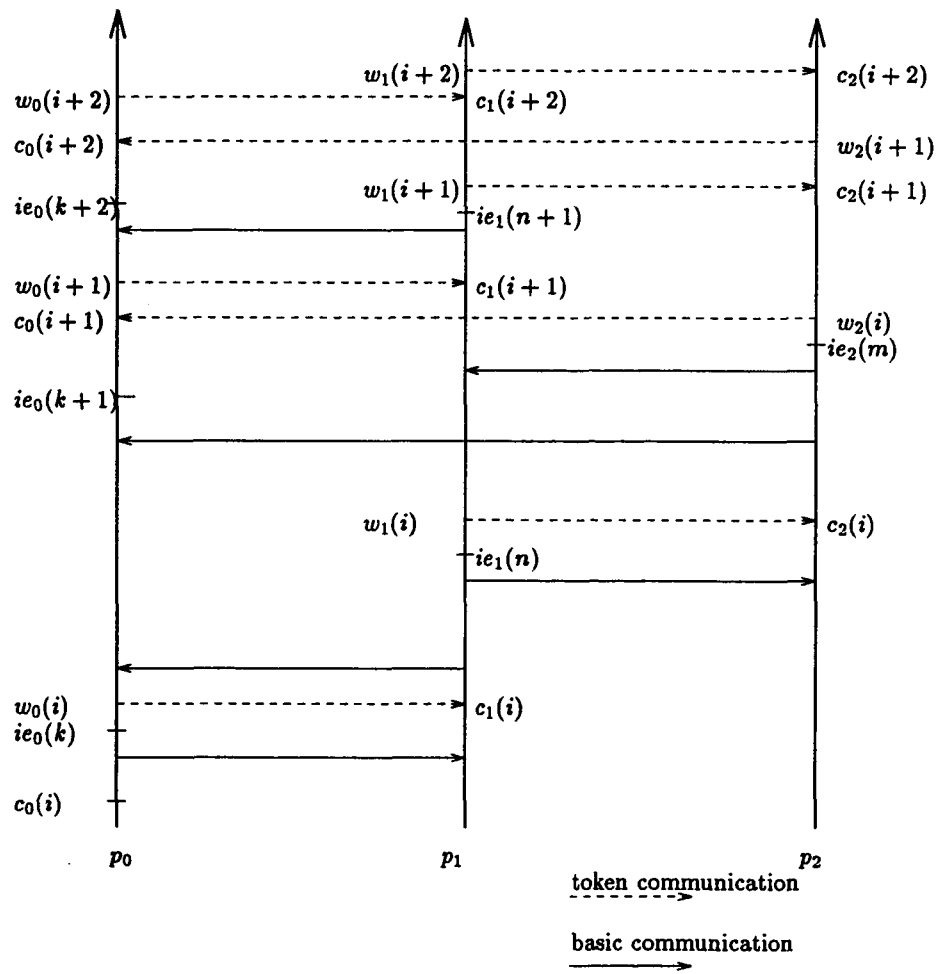


Figure 2.1: Control Wave Events

We also define a predicate $Idle()$ on events which corresponds to the state of a process at an event (independent of the termination detection protocol). The function $Idle(e') = False$ if e' is a send or receive event. For any other event e' on p_j , $Idle(e') = False$ iff (there does not exist $\eta(s)$ such that $\rho(s) = p_j \wedge \eta(s) \rightarrow e'$), and (there does not exist $ie_j(k)$ such that $\eta(s) \rightarrow ie_j(k) \rightarrow e'$). Figure 2.1 shows an example of the control wave and idle events.

Given this predicate on events, we can specify that termination of a computation composed of processes $p_j \in \Pi$ has occurred at the completion of a polling wave $PW(i)$ when the following conditions hold:

Termination Conditions - T

T(a) $Idle(w_j(i))$ for all $w_j(i) \in PW(i)$; and

T(b) For all receive events $\eta(s)$ in the computation there exists $w_{\rho(s)}(i) \in PW(i)$ such that $\eta(s) \rightarrow w_{\rho(s)}(i)$.

For a termination detection algorithm of this type to be correct, any wave completed during execution of the protocol for which the above conditions do not hold should be considered invalid, or failed. A successful, or final wave, $PW(F)$, would be one for which those conditions are true.

In each of the polling protocols we have discussed a process is marked if it becomes active. This mark is erased by the departure of the control message. We define the function $Pstat()$ to correspond to the marking of a process. $Pstat()$ is a function defined on the set of c events. The value of $Pstat(c_j(i))$ is false if p_j has been active since the occurrence of $c_j(i - 1)$. $Pstat(c_j(i))$ is true otherwise.

The following rules define the standard paradigm for detecting termination in an asynchronous system using synchronous communication. The information contained in the control message of polling wave $PW(i)$ is signified by $tk(i)$.

Asynchronous System with Synchronous Communication - Generic Protocol

G.1 $\neg Pstat(c_j(i))$ iff

$$\begin{aligned}
 & i = 1 \vee \\
 & (i > 1) \wedge \exists \eta(s) \text{ such that } w_{\rho(s)}(i-1) \rightarrow \eta(s) \rightarrow c_{\rho(s)}(i) \vee \\
 & (i > 1) \wedge \exists s \text{ such that } w_{\sigma(s)}(i-1) \rightarrow s \rightarrow c_{\sigma(s)}(i).
 \end{aligned}$$

G.2 The occurrence of $w_j(i)$ implies

$$\begin{aligned}
 & Idle(c_j(i)) \wedge c_j(i) \mapsto w_j(i) \vee \\
 & \left[\begin{array}{l} \neg Idle(c_j(i)) \wedge c_j(i) \rightarrow ie_j(k) \mapsto w_j(i) \wedge \\ \text{if } c_j(i) \rightarrow e'_j \rightarrow ie_j(k) \text{ then } e'_j \text{ is a send or receive.} \end{array} \right]
 \end{aligned}$$

G.3 $\neg Pstat(c_j(i))$ implies $tk(i) = active$.

G.4 A polling wave, $PW(i)$ is valid iff $c_0(i)$ has occurred, and $tk(i) = idle$.

G.5 Event $w_0(i+1)$ occurs iff

$$\begin{aligned}
 & c_0(i) \text{ occurs} \wedge \\
 & tk(i) = active \wedge \\
 & tk(i+1) = idle.
 \end{aligned}$$

Now we need to show that this standard algorithm satisfies Termination Conditions T(a) and T(b); that is, a valid wave of this protocol guarantees that T holds. T(a) requires that $Idle(w_j(i))$ for all $w_j(i) \in PW(i)$. Based on the specifications for the occurrence of $w_j(i)$ given in Rule G.2, it is easily shown that this condition is met in every polling wave, including the valid final wave.

Lemma 1 *For any $w_j(i)$ event as it is specified in the Asynchronous System and Synchronous Communication - Generic Protocol, $Idle(w_j(i))$.*

Proof: Assume that there exists $w_j(i)$ such that $\neg Idle(w_j(i))$. If this were the case, then there exists $\eta(s)$ such that $\rho(s) = p_j$, and $\eta(s) \rightarrow w_j(i)$. In addition there would *not* exist an idle event $ie_j(k)$ such that $\eta(s) \rightarrow ie_j(k) \rightarrow w_j(i)$. This directly contradicts Rule G.2 which requires that there be a $ie_j(k)$ such that $ie_j(k) \mapsto w_j(i)$. $\neg Idle(w_j(i))$ also contradicts Rule G.2. If $Idle(c_j(i))$ then there exists $ie_j(k)$, such that $ie_j(k) \rightarrow c_j(i)$ without an intervening receive or send, and $c_j(i) \rightarrow w_j(i)$ without an intervening event. Hence, $ie_j(k) \rightarrow w_j(i)$, and there is no $\eta(s)$ such that $ie_j(k) \rightarrow \eta(s) \rightarrow w_j(i)$. ■

To determine whether T(b) holds, we need to show that if polling wave $PW(i)$ is valid in this protocol then there is no receive event $\eta(s)$ in the underlying computation such that $w_j(i) \rightarrow \eta(s)$ for any $w_j(i) \in PW(i)$. First, we define an *inter-wave interval* set $IW(i)$, for $i > 1$. A send event $s \in IW(i)$ iff $w_{\sigma(s)}(i-1) \rightarrow s \rightarrow w_{\sigma(s)}(i)$. Similarly, a receive event $\eta(s) \in IW(i)$ iff $w_{\rho(s)}(i-1) \rightarrow \eta(s) \rightarrow w_{\rho(s)}(i)$. ■

Lemma 2 *If a polling wave $PW(i)$ is valid, then there are no send or receive events that are elements of $IW(i)$.*

Proof: Assume polling wave $PW(i)$ is valid. Rule G.4 requires that if $PW(i)$ is valid then $c_0(i)$ has occurred, and $tk(i) = idle$. If $c_0(i)$ has occurred then every process has been polled during the i^{th} wave. Therefore, by Rule G.3, $Pstat(c_j(i))$ at $c_j(i)$ for all $c_j(i) \in PW(i)$. Rule G.1 requires that $\neg Pstat(c_j(i))$ if there exists a send s such that $\sigma(s) = p_j$ and $w_j(i-1) \rightarrow s \rightarrow c_j(i)$, or a receive $\eta(s)$ such that $\rho(s) = p_j$ and $w_j(i-1) \rightarrow \eta(s) \rightarrow c_j(i)$. So any communication which occurs after $w_j(i-1)$ and before $c_j(i)$, for any $p_j \in \Pi$, will invalidate $PW(i)$, contradicting our original assumption. $Pstat(c_j(i))$ also implies that $Idle(c_j(i))$. By Rule G.2, if $Idle(c_j(i))$ then there is no event e'_j such that $c_j(i) \rightarrow e'_j \rightarrow w_j(i)$. Therefore, there can be no communication events which occur after $w_j(i)$ and before $w_j(i)$. ■

The requirement of synchronous communication has a significant impact on the causal relationships between events. In a system with asynchronous message passing, a send “happens before” its receipt, so that $s \rightarrow \eta(s)$. This is not the case when communication is synchronous; the action of sending a message and receiving a message in this regime may be viewed as one event that links two processes temporarily. Associating different names with the send and receive is a notational convenience. Both names signify the same event, and for this reason it no longer makes sense to say that the send “happens before” the receive.

By temporarily linking two processes, synchronous communication creates stronger causal relationships between events in the communicating processes than asynchronous message passing. Transitivity of the \rightarrow relation is used to establish a causal relationship between any event that precedes an asynchronous send in one process and the events which follow the corresponding receive. Asynchronous communication does not create a causal relationship between events in the receiving process which precede the receive and events in the sending process which follow the send. However, when communication is synchronous a causal relationship is established between events which happen before the receive in the receiving process and events which causally follow the send in the sending process. In other words, transitivity holds “in both directions” when synchronous communication occurs.

The effects of synchronous communication on causality are summarized below.

Rules for Causality under Synchronous Communication

- $s \not\rightarrow \eta(s)$, and $\eta(s) \not\rightarrow s$.
 - For any event e'_i such that $e'_i \rightarrow \eta(s)$, it is true that $e'_i \rightarrow s$.
 - For any event e'_i such that $s \rightarrow e'_i$, and event e'_j such that $e'_j \rightarrow \eta(s)$, it is true that $e'_j \rightarrow e'_i$.
-

These characteristics of synchronous communication provide the leverage needed to make the protocols described in this section function correctly. They also establish the causal relationships needed to complete the proof that the protocol meets the required termination conditions. Lemma 3 shows how this generic protocol and those in [14, 18, 15, 17] require synchronous communication for their correct operation.

Lemma 3 *If polling wave, $PW(i)$, as specified in the Generic Protocol is valid then Termination Condition $T(b)$ is satisfied.*

Proof: Assume $PW(i)$ is successful yet there exists a receive $\eta(s)$ such that $\rho(s) = p_j$, and $w_j(i) \rightarrow \eta(s)$. Because $w_k(i-1) \rightarrow w_k(i)$ for all $w_k(i-1) \in PW(i-1)$, $w_k(i-1) \rightarrow \eta(s)$ for all $w_k(i-1) \in PW(i-1)$. Synchronous communication requires that if $w_k(i-1) \rightarrow \eta(s)$ then $w_k(i-1) \rightarrow s$, for all $w_k(i-1) \in PW(i-1)$. By Lemma 2 the send s may not be in $IW(i)$, so if $w_{\sigma(s)}(i-1) \rightarrow s$ then $w_{\sigma(s)}(i) \rightarrow s$. However, by Lemma 1, for all $p_k \in \Pi$, p_k is idle at $w_k(i)$, and by definition, an idle process may not send a message. ■

Theorem 1 *The completion of a valid wave in the Generic Protocol satisfies Termination Conditions $T(a)$ and $T(b)$.*

Proof: Follows directly from Lemmas 1 and 3. ■

The following lemma shows that the protocol will detect termination if it has occurred.

Lemma 4 *If termination conditions $T(a)$ and $T(b)$ hold then there exists i such that $PW(i)$ of the Generic Protocol is a valid wave.*

Proof: By Rule G.5, successive polling waves will be instigated until a valid wave occurs. Therefore, given that $T(a)$ holds, there will be a polling wave $PW(k)$ for which $Idle(c_j(k))$ will hold for all $p_j \in \Pi$. Upon completion of $PW(k)$, the value of $tk(k)$ may be idle or active depending on the activity of the process before the polling events of $PW(k)$. If $tk(k) = idle$

then, by Rule G.4, $PW(k)$ is a valid wave, and by Rule G.5, the termination detection protocol terminates. If $tk(k) = active$ then, by Rule G.5, $w_0(k+1)$ will occur, and $tk(k+1)$ will be set to idle. Since the underlying computation is terminated, $Idle(c_j(k+1))$ will hold for all $p_j \in \Pi$. Termination condition $T(b)$ holds for $PW(k)$, therefore, $Pstat(c_j(k+1))$ (Rule G.1) for all $p_j \in \Pi$. By Rule G.3, $tk(k+1)$ will not be changed during the wave, and upon the occurrence of $c_0(k+1)$ the valid wave $PW(k+1)$ will have completed. ■

2.2 Fully Synchronous Systems

Rana [16] proposes an algorithm that is suitable for a system environment which supports synchronous communications and synchronized clocks. The protocol requires that a global time value can be associated with any event that occurs at a process. This capability is used to verify that all processes are idle simultaneously.

As in Dijkstra's termination detection algorithm, Rana uses a token circulating in a virtual ring to generate a control wave. Rana's algorithm differs in that more than one process may originate a control wave. In this protocol any process, upon going idle, records the time at which it went idle and generates a token with its identification number and time the process went idle. An active process receiving the incoming token discards it. Similarly, an idle process with a "went idle" time greater than the timestamp in the incoming token also discards the token. A process only propagates an incoming token if it is idle and the timestamp of the token exceeds the process' latest idle time. A token returning to its originator indicates termination.

In this algorithm several tokens may be active concurrently, and more than one control wave may be identified. However, only one token will return to its originator, and only one control wave will be completed. For this reason we modify the notation we developed in the previous section as necessary.

- $c_j(i, n)$ signifies the arrival of the n^{th} token generated by p_i at process p_j .

- $w_j(i, n)$ represents the event of the n^{th} token generated by p_i leaving p_j .
- $PW(i, n)$ will signify the polling wave created by the n^{th} token from process p_i .
- $tk(i, n)$ signifies the n^{th} token generated by p_i and $tk(i, n).ts$ is the timestamp in that token.
- $T(e)$ is a function which returns the physical time of an event's occurrence.

Note that the i in $w_j(i, n)$, or $c_j(i, n)$, no longer indicates that this is a control event of the i^{th} wave, rather it signifies that process i originated the wave. Also note that more than one token from a process may exist in the system concurrently. For this reason it is necessary to associate a token index number with the token, the wave events and each polling wave.

Fully Synchronous System Protocol - Rana

R.1 $w_j(i, m), i \neq j$ occurs iff

$$\begin{aligned}
 & Idle(c_j(i, m)) \wedge \\
 & T(ie_j(k)) < tk(i, m).ts \wedge \\
 & \nexists ie_j(n) \text{ such that } ie_j(k) \rightarrow ie_j(n) \rightarrow c_j(i, m) \wedge \\
 & c_j(i, m) \mapsto w_j(i, m).
 \end{aligned}$$

R.2 $w_j(j, m)$ occurs iff $\exists ie_j(k)$ such that $ie_j(k) \mapsto w_j(j, m)$.

R.3 The occurrence of $w_j(j, m)$ implies $tk(j, m).ts = T(ie_j(k))$, where $ie_j(k) \mapsto w_j(j, m)$.

R.4 A polling wave, $PW(j, m)$ is valid when $c_j(j, m)$ occurs for some $p_j \in \Pi$.

Lemma 5 *For any $w_j(i, m)$ event as it is specified in Rana's Protocol, $Idle(w_j(i, m)) = True$.*

Proof: Rule R.1 requires that $Idle(w_j(i, m))$ be true. In the case of the generation of a token at $w_j(j, m)$, clearly Rule R.2 guarantees that $Idle(w_j(j, m))$ is true as well. ■

The following rules summarize the relationship between real time, causality, and synchronous communication:

Rules for Causality and Time in a Fully Synchronous System

- For any events e'_i and e'_j , if $e'_i \rightarrow e'_j$ then $T(e'_i) < T(e'_j)$.
 - For send and receive events s and $\eta(s)$, $T(s) = T(\eta(s))$.
-

These properties make the proof that Rana's protocol satisfies Termination Condition T(b) straightforward.

Lemma 6 *If polling wave, $PW(i, n)$, as specified in Rana's Protocol is valid then Termination Condition T(b) is satisfied.*

Proof: If T(b) does not hold then there exists $\eta(s)$ such that $w_{\rho(s)}(i, n) \rightarrow \eta(s)$. There are two possible cases, either $w_{\rho(s)}(i, n) \rightarrow s \rightarrow w_{\sigma(s)}(i, n)$, or $w_{\sigma(s)}(i, n) \rightarrow s$. By Lemma 5, $p_{\sigma(s)}$ is idle at $w_{\sigma(s)}(i, n)$, so it is not possible that $w_{\sigma(s)}(i, n) \rightarrow s$. If $s \rightarrow w_{\sigma(s)}(i, n)$ then $s \rightarrow ie_{\sigma(s)}(k) \rightarrow w_{\sigma(s)}(i, n)$ for some idle event $ie_{\sigma(s)}(k)$, and $T(s) < T(ie_{\sigma(s)}(k)) < T(w_{\sigma(s)}(i, n))$. Because communication is synchronous, $T(\eta(s)) < T(ie_{\sigma(s)}(k))$ as well. $PW(i, n)$ is a valid wave, so by Rule R.1, $tk(i, n).ts \geq T(ie_{\sigma(s)}(k))$ for any $ie_{\sigma(s)}(k) \rightarrow w_{\sigma(s)}(i, n)$. Therefore, $tk(i, n).ts > T(\eta(s))$, $T(w_{\rho(s)}(i, n)) > tk(i, n).ts$, and $T(w_{\rho(s)}(i, n)) > T(\eta(s))$. This implies $\eta(s) \rightarrow w_{\rho(s)}(i, n)$, thus contradicting our original assumptions. ■

Theorem 2 *The completion of a valid wave in Rana's protocol satisfies Termination Conditions $T(a)$ and $T(b)$.*

Proof: Follows directly from Lemmas 5 and 6. ■

Lemma 7 *If termination conditions $T(a)$ and $T(b)$ hold then there exists i such that $PW(i)$ of Rana's Protocol is a valid wave.*

Proof: By Rule R.2, a control wave is instigated by every idle event. There will necessarily be a latest idle event. Let $ie_j(k)$ be the latest idle event. By Rule R.3, $tk(j, m).ts = T(ie_j(k))$. Since $ie_j(k)$ is the latest idle event, $tk(j, m).ts > T(ie_l(m))$ for all $l \neq j$. Therefore, by Rule R.1, $w_l(j, k)$ will occur for every $p_l \in \Pi, l \neq j$, and $c_j(j, k)$ will occur completing a valid wave. ■

2.3 Asynchronous Systems and Asynchronous Communication

2.3.1 First-In, First-Out Channel Restrictions

Synchronous systems and synchronous message passing are fairly expensive and unrealistic restrictions. Misra proposes two algorithms which relax these assumptions [20]. Asynchronous communication is assumed, however message ordering in the channels (FIFO channels) is required.

An extremely simple protocol is suitable when the processes are arranged in a ring. In this case the control wave takes the form of a token or marker which traverses the ring painting idle processes white. A process colors itself black if it becomes active. If the token arrives at a white process it knows that the process has been continuously idle since the last visit. Termination can be declared if N processes in a row are found to be white by the token. The ring topology and the use of a token satisfies our second condition for

termination. FIFO channels guarantee that any message sent before a wave will arrive at its destination before the wave does.

The situation is more complicated for arbitrary networks. In the protocol applicable to the more general case, the marker must traverse all the communication links to flush out any message in transit. Such a traversal can be accomplished by preprocessing the network to determine an appropriate cycle that includes every edge, or by instituting a depth first search to cover all links.

In this protocol every process is initially black. An arbitrary node initiates the procedure when it becomes idle by painting itself white, setting a counter to one, and sending the token out on an outgoing edge of the cycle. A process which receives the token passes it on when it becomes idle. If that process is white it increments the counter in the token by one. If it is black it resets the counter to 0 before propagating it and paints itself white. Processes turn black when they receive a message. Termination is declared when M white processes have been visited in a row, where M is the number of edges in the cycle.

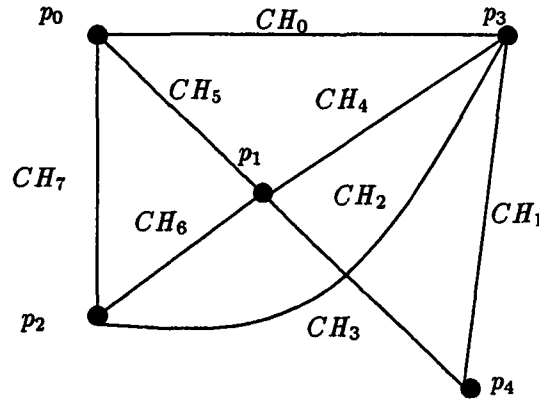
The system in which this protocol operates is complicated by the lack of synchronous communication, but the requirement of FIFO channels allows the protocol to behave in a way similar to the synchronous communication algorithms. To show this we will restate Misra's algorithm using the terminology we developed earlier. First we must define the token behavior.

Let the system be composed of N processors and M communication channels. The system must be preprocessed to determine a cycle of edges which, when followed by the token, will insure that each channel is traversed at least once. Each of these edges can be uniquely defined as a triple; (outgoing process, channel, incoming process). The outgoing process signifies the process the token is leaving. The incoming process is the token's destination. Let Φ be an ordered set whose elements are in one-to-one correspondence to the triples representing the edges of the cycle. Let PT be the cardinality of this set, and $\Phi = \{path_0, path_1, \dots, path_{PT-1}\}$, where $path_i$ is the i^{th} link in the token's specified

traversal.

Define the following functions:

1. $F(p_i, Channel_m, p_j) = path_k$ where $F()$ defines the correspondence between the edge triple in the traversal graph and the appropriate element of Φ .
2. $Pred(path_k) = p_i$ where p_i is the outgoing process.
3. $Succ(path_k) = p_j$ where p_j is the incoming process.



$$\Phi = \{(p_0, CH_0, p_3), \\ (p_3, CH_1, p_4), \\ (p_4, CH_3, p_1), \\ (p_1, CH_4, p_3), \\ (p_3, CH_2, p_2), \\ (p_2, CH_6, p_1), \\ (p_1, CH_5, p_0), \\ (p_0, CH_7, p_2)\}$$

Figure 2.2: Construction of Φ set

Figure 2.2 illustrates how the Φ set is constructed for a sample system. In this example $path_0 = (p_0, CH_0, p_3)$, $path_1 = (p_3, CH_1, p_4)$, etc. $Pred(path_0) = p_0$ and $Succ(path_0) = p_3$. $Pred(path_1) = p_3$ and $Succ(path_1) = p_4$. The outgoing process of two different paths may be the same. For example, $Pred(path_0) = Pred(path_7) = p_0$. The incoming process of

different paths may also be the same. Consequently, a token may traverse a process more than once.

The wave events must be redefined to correspond to the token's behavior. Let $c_j(i)$ signify the event of a control message of the i^{th} polling wave arriving at a process on $path_j$. The event of a control message leaving a process on the i^{th} wave on $path_j$ will be indicated by $w_j(i)$. A polling wave $PW(i)$ is composed of $C(i) = \{c_0(i), c_1(i), \dots, c_{PT-1}(i)\}$ and $W(i) = \{w_0(i), w_1(i), \dots, w_{PT-1}(i)\}$.

In Misra's algorithm, as in Rana's, it is possible for any process to be the initiator of the final wave. At first glance it appears that the protocol is composed of numerous abortive waves and one complete and final wave. In actuality the principle of operation is the same as those proposed by Dijkstra and Francez: the processes are repeatedly polled until they are found to be continuously idle. Allowing any process to recognize termination simply eliminates the inefficiency of returning the markers or other control messages to a specific process. For this reason, and for ease of exposition, we will assume that process p_0 is the initiator of the final wave. Given this assumption we can define an inter-wave interval set and show that this algorithm satisfies termination conditions in a manner analogous to our previous proofs.

Asynchronous System with FIFO Communications Protocol - Misra

MI.1 $\neg Pstat(c_j(i))$ iff

$$i = 1 \vee \left[\begin{array}{l} (i > 1) \wedge \\ \exists \eta(s) \text{ such that } \rho(s) = p_k \wedge \\ Succ(path_j) = Pred(path_{j+1}) = p_k \wedge \\ w_{j+1}(i-1) \rightarrow \eta(s) \rightarrow c_j(i) \\ (i > 1) \wedge \\ \exists s \text{ such that } \sigma(s) = p_k \wedge \\ Succ(path_j) = Pred(path_{j+1}) = p_k \wedge \\ w_{j+1}(i-1) \rightarrow s \rightarrow c_j(i). \end{array} \right] \vee$$

MI.2 The occurrence of $w_j(i)$ implies

$$\left[\begin{array}{l} Idle(c_{j-1}(i)) \wedge \\ c_{j-1}(i) \mapsto w_j(i) \wedge \\ Succ(path_{j-1}) = Pred(path_j) = p_k \end{array} \right] \vee \left[\begin{array}{l} \neg Idle(c_{j-1}(i)) \wedge \\ Succ(path_{j-1}) = Pred(path_j) = p_k \wedge \\ c_{j-1}(i) \rightarrow ie_m(k) \mapsto w_j(i) \wedge \\ c_{j-1}(i) \rightarrow e'_k \rightarrow ie_m(k) \text{ implies } e'_k \text{ is a send or receive.} \end{array} \right]$$

MI.3 $\neg Pstat(c_j(i))$ implies $tk(i) = active$.

MI.4 A polling wave, $PW(i)$ is valid iff $c_0(i)$ has occurred, and $tk(i) = idle$.

MI.5 Event $w_0(i+1)$ occurs iff

$$\begin{array}{l} c_0(i) \text{ occurs } \wedge \\ tk(i) = active \wedge \\ tk(i+1) = idle. \end{array}$$

The inter-wave interval set for this protocol is defined as follows. A send event $s \in IW^*(i)$ if $\sigma(s) = p_k$, $Pred(path_j) = p_k$, and $w_j(i-1) \rightarrow s \rightarrow w_j(i)$, for some $w_j(i) \in PW(i)$. A receive event $\eta(s) \in IW(i)^*$ if $\rho(s) = p_k$, $Pred(w_j(i)) = p_k$, and $w_j(i-1) \rightarrow \eta(s) \rightarrow w_j(i)$.

Lemma 8 *If a polling wave $PW(i)$ is valid, then there are no send or receive events that are elements of $IW(i)^*$*

Proof: Assume polling wave $PW(i)$ is valid. Rule MI.4 requires that if $PW(i)$ is valid then $c_0(i)$ has occurred, and $tk(i) = idle$. If $c_0(i)$ has occurred then every path in Φ has been polled during the i^{th} wave. Therefore, by Rules MI.3 and MI.4, $Pstat(c_j(i))$ at $c_j(i)$ for all $c_j(i) \in PW(i)$. By Rule MI.1 of the protocol, any send s such that $\sigma(s) = p_j$, $Succ(path_{k-1}) = Pred(path_k) = p_j$, and $w_k(i-1) \rightarrow s \rightarrow c_{k-1}(i)$ would cause $\neg Pstat(c_{k-1}(i))$. The same would be true of a receive. Therefore any communication occurring between $w_k(i-1)$ and $c_{k-1}(i)$ would invalidate $PW(i)$, contradicting our original assumption. $Pstat(c_k(i))$ implies that $Idle(c_j(i))$. By Rule MI.2 if $Idle(c_j(i))$ is true then there does not exist event e'_j such that $c_{k-1}(i) \rightarrow e'_j \rightarrow w_k(i)$. Therefore, there can be no communication events which occur after $w_k(i-1)$ and before $w_k(i)$. ■

Note there is very little difference between Lemmas 2 and 8 or their proofs. Lemma 8 will be used to show that this protocol satisfies termination condition T(b), as Lemma 2 was used for the generic protocol. First we show that Misra's protocol satisfies termination condition T(a).

Lemma 9 *For any $w_j(i)$ event as it is specified in the Misra Protocol, $\text{Idle}(w_j(i))$.*

Proof: Assume that there exists $w_j(i)$ such that $\neg \text{Idle}(w_j(i))$. If this were the case, then there exists $\eta(s)$ such that $\rho(s) = p_m$, $\eta(s) \rightarrow w_j(i)$, and $\text{Pred}(\text{path}_j) = p_m$. In addition there would not exist an idle event $ie_m(k)$ such that $\eta(s) \rightarrow ie_m(k) \rightarrow w_j(i)$. This directly contradicts Rule MI.2 which requires that there exists $ie_m(k)$ such that $ie_m(k) \rightarrow w_j(i)$, and there does not exist e'_j such that $ie_m(k) \rightarrow e'_j \rightarrow w_j(i)$. $\neg \text{Idle}(w_j(i))$ also contradicts Rule MI.2. If $\text{Idle}(c_{j-1}(i))$ as specified in Rule MI.2, then there exists $ie_m(k)$ such that $ie_m(k) \rightarrow c_{j-1}(i)$ without an intervening receive, and $c_{j-1}(i) \rightarrow w_j(i)$ without an intervening event. Hence $ie_m(k) \rightarrow w_j(i)$ and there does not exist $\eta(s)$ such that $ie_m(k) \rightarrow \eta(s) \rightarrow w_j(i)$. ■

Lemma 10 *If polling wave, $PW(i)$, as specified in Misra Protocol is valid then Termination Condition T(b) is satisfied.*

Proof: Assume $PW(i)$ is valid yet there exists a receive $\eta(s)$ that violates T(b), so that $\rho(s) = p_j$, $\text{Pred}(\text{path}_k) = p_j$, and $w_k(i) \rightarrow \eta(s)$. Let $\eta(s)$ be the *earliest* such receive. By Lemma 8 the corresponding send must have originated before the wave, so that $s \rightarrow w_j(i)$ for some $w_j(i) \in PW(i)$. Let $\sigma(s) = p_k$, and the path corresponding to the channel on which s is sent and the process p_k , be path_m . By Lemma 8, $s \notin IW(i)$, so $s \rightarrow w_j(i-1)$ for all $w_j(i-1) \in PW(i)$ such that $\text{Pred}(\text{path}_j) = p_k$. In particular, $s \rightarrow w_m(i-1)$. Because communication is FIFO, $\eta(s) \rightarrow c_m(i-1)$. We also know that $c_m(i-1) \rightarrow w_{m+1}(i-1)$, and $w_{m+1}(i-1) \rightarrow w_j(i)$ for all $w_j(i) \in PW(i)$. Hence, $\eta(s) \rightarrow w_j(i)$ for all $w_j(i) \in PW(i)$. This contradicts our initial assumption that $w_j(i) \rightarrow \eta(s)$ for some $w_j(i) \in PW(i)$. ■

Theorem 3 *The completion of a valid wave in the Misra Protocol satisfies Termination Conditions $T(a)$ and $T(b)$.*

Proof: Follows directly from Lemmas 9 and 10. ■

Lemma 11 *If termination conditions $T(a)$ and $T(b)$ hold then there exists i such that $PW(i)$ of the Misra Protocol is a valid wave.*

Proof: By Rule MI.5 successive polling waves will be instigated until a valid wave occurs. Therefore, given that $T(a)$ holds, there will be a polling wave $PW(k)$ for which $Idle(c_j(k))$ will hold for all $c_j(k) \in PW(k)$. Upon completion of $PW(k)$, the value of $tk(k)$ may be idle or active depending on the activity of the process before the polling events of $PW(k)$. If $tk(k) = idle$ then, by Rule MI.4, $PW(k)$ is a valid wave, and by Rule MI.5, the termination detection protocol terminates. If $tk(k) = active$ then, by Rule MI.5, $w_0(k+1)$ will occur, and $tk(k+1)$ will be set to idle. Since the underlying computation is terminated, $Idle(c_j(k+1))$ will hold for all $c_j(k+1) \in PW(k+1)$. Termination condition $T(b)$ holds for $PW(k)$, therefore, $Pstat(c_j(k+1))$ (Rule MI.1) for all $c_j(k+1) \in PW(k+1)$. By Rule MI.3 $tk(k+1)$ will not be changed during the wave, and upon the occurrence of $c_0(k+1)$ the valid wave $PW(k+1)$ will have completed. ■

2.3.2 Fully Asynchronous Systems

In an asynchronous system with totally asynchronous message transmission in which FIFO ordering is not enforced, it is difficult to assert that all messages sent will be received before a successful control wave. In the previous algorithms we have discussed we have seen how synchronous communication guaranteed that a message sent before one control wave would be received before the next wave. In Misra's algorithms FIFO channels were used to guarantee that the control wave arrived after the receipt of any message sent before the control wave. In an asynchronous system with true asynchronous communication there is no

similar mechanism. The only existing algorithms applicable to such systems are suggested by Mattern[19]. These algorithms verify termination by counting the number of messages sent and received by all processes.

The first of these algorithms, which he calls the Four Counter Algorithm, operates by propagating a control wave through the processes in a manner similar to those protocols we have already discussed. Instead of querying the process to determine if it has been active, the purpose of the control wave is to poll each process for the number of messages it has sent and received. These values are accumulated in a token as $tk(i).IN_1$ and $tk(i).OUT_1$. After completion of the first control wave, a second wave is initiated which performs the same function and accumulates messages received and sent into $tk(i).IN_2$ and $tk(i).OUT_2$. The second wave is considered successful if $tk(i).IN_1 = tk(i).OUT_1 = tk(i).IN_2 = tk(i).OUT_2$. If the values do not balance then termination has not occurred. In this case $tk(i).IN_1$ is set equal to $tk(i).IN_2$, $tk(i).OUT_1$ is set to $tk(i).OUT_2$, and a new control wave is initiated to obtain new values for $tk(i).IN_2$ and $tk(i).OUT_2$. To define this algorithm in the terminology we have used thus far, we need to add variables used for accumulating message counts. For all $p_j \in \Pi$, in_j and out_j accumulate the number of messages received and sent by a process p_j .

Fully Asynchronous System - Mattern's Four Counter Protocol

MAC.1 A send s such that $\sigma(s) = p_j$, sets $out_j = out_j + 1$.

MAC.2 A receive $\eta(s)$ such that $\rho(s) = p_j$, sets $in_j = in_j + 1$.

MAC.3 The occurrence of $w_j(i)$ implies

$$\begin{aligned} & Idle(c_j(i)) \wedge c_j(i) \mapsto w_j(i) \vee \\ & \left[\begin{array}{l} \neg Idle(c_j(i)) \wedge c_j(i) \rightarrow ie_j(k) \mapsto w_j(i) \wedge \\ c_j(i) \rightarrow e'_j \rightarrow ie_j(k) \text{ implies } e'_j \text{ is a send or receive.} \end{array} \right] \end{aligned}$$

MAC.4 The occurrence of $w_j(i), i \geq 2$ implies

$$\begin{aligned} tk(i).IN_2 &= tk(i).IN_2 + in_j \wedge \\ tk(i).OUT_2 &= tk(i).OUT_2 + out_j. \end{aligned}$$

MAC.5 The occurrence of $w_j(i), i = 1$ implies

$$\begin{aligned} tk(i).IN_1 &= tk(i).IN_1 + in_j \wedge \\ tk(i).OUT_1 &= tk(i).OUT_1 + out_j. \end{aligned}$$

MAC.6 A polling wave, $PW(i)$ is valid iff

$$\begin{aligned} c_0(i) \text{ has occurred} \wedge \\ tk(i).IN_1 = tk(i).OUT_1 = tk(i).IN_2 = tk(i).OUT_2 \wedge \\ i > 1. \end{aligned}$$

MAC.7 Event $w_0(i + 1)$ occurs iff

$$\begin{aligned} c_0(i) \text{ occurs} \wedge \\ PW(i) \text{ is not valid} \wedge \\ tk(i + 1).IN_1 = tk(i).IN_2 \wedge \\ tk(i + 1).OUT_1 = tk(i).OUT_2 \wedge \\ tk(i + 1).OUT_2 = tk(i + 1).IN_2 = 0. \end{aligned}$$

MAC.8 Initialization:

$$\begin{aligned} tk(0).IN_1 &= tk(0).OUT_1 = 0, \\ tk(0).IN_2 &= tk(0).OUT_2 = 0, \\ in_j &= out_j = 0 \text{ for all } p_j \in \Pi. \end{aligned}$$

Prior to showing that this algorithm satisfies our termination conditions we will prove the following lemma. Note that it is essentially a restatement of Lemma 2 as it applies to this protocol.

Lemma 12 *If a polling wave $PW(i)$ is valid in the Mattern's Four Counter Protocol, then there is no send or receive in $IW(i)$.*

Proof: Assume a polling wave, $PW(i)$ is valid. Let s be a send such that $\sigma(s) = p_j$, and $s \in IW(i)$. According to the definition of $IW(i)$, this implies that $w_j(i-1) \rightarrow s \rightarrow w_j(i)$. By Rules MAC.1 and MAC.4, the value of out_j at $w_j(i)$ must exceed the value of out_j at $w_j(i-1)$ by at least one, so $tk(i).OUT_2 \geq tk(i).OUT_1 + 1$ at $w_j(i)$, and at any $w_k(i)$ such that $w_j(i) \rightarrow w_k(i)$. This contradicts our assumption that $PW(i)$ is valid. A similar argument can be made in the case that a receive is assumed to be in $IW(i)$. ■

Lemma 13 *For any $w_j(i)$ event as it is specified in Mattern's Four Counter Protocol, $Idle(w_j(i))$.*

Proof: Assume that there exists $w_j(i)$ such that $\neg Idle(w_j(i))$. If this were the case, then there exists $\eta(s)$ such that $\rho(s) = p_j$, and $\eta(s) \rightarrow w_j(i)$. In addition there would *not* exist an idle event $ie_j(k)$ such that $\eta(s) \rightarrow ie_j(k) \rightarrow w_j(i)$. This directly contradicts Rule MAC.3 which requires that there be a $ie_j(k)$ such that $ie_j(k) \mapsto w_j(i)$. $\neg Idle(w_j(i))$ also contradicts Rule MAC.3. If $Idle(c_j(i))$ then there exists $ie_j(k)$, such that $ie_j(k) \rightarrow c_j(i)$ without an intervening receive or send, and $c_j(i) \rightarrow w_j(i)$ without an intervening event. Hence, $ie_j(k) \rightarrow w_j(i)$, and there is no $\eta(s)$ such that $ie_j(k) \rightarrow \eta(s) \rightarrow w_j(i)$. ■

Lemma 14 *If polling wave, $PW(i)$, as specified in Mattern's Four Counter Protocol is valid then Termination Condition $T(b)$ is satisfied.*

Proof: Assume that polling wave $PW(i)$ is valid, and there exists $\eta(s)$ such that $\rho(s) = p_j$, and $w_j(i) \rightarrow \eta(s)$. Let s be the corresponding send and $\sigma(s) = p_k$. Also let $\eta(s)$ be

the earliest such receive. By Lemma 13, $s \rightarrow w_k(i)$. By Lemma 12, $s \notin IW(i)$, hence $s \rightarrow w_k(i-1)$. If this is true, $tk(i).OUT_1$ and $tk(i).OUT_2$ include a count for s , but $tk(i).IN_1$ and $tk(i).IN_2$ do not include a count for the corresponding receive. However, by assumption $tk(i).OUT_2 = tk(i).OUT_1 = tk(i).IN_2 = tk(i).IN_1$. This can only occur if there exists s^* and $\eta(s^*)$ such that $\sigma(s) = p_g$, and $\rho(s^*) = p_m$ for $p_g, p_m \in \Pi$, and $\eta(s^*) \rightarrow w_m(i-1)$, and $w_g(i) \rightarrow s^*$. Because $w_m(i-1) \rightarrow w_j(i)$ for all $w_j(i) \in PW(i)$, we have the following contradiction: $\eta(s^*) \rightarrow w_m(i-1) \rightarrow w_k(i) \rightarrow s^* \rightarrow \eta(s^*)$. ■

Theorem 4 *The completion of a valid wave in the Mattern's Four Counter Protocol satisfies Termination Conditions T(a) and T(b).*

Proof: Follows directly from Lemmas 13 and 14 ■

Lemma 15 *If termination conditions T(a) and T(b) hold then there exists i such that $PW(i)$ in Mattern's Four Counter Protocol is a valid wave.*

Proof: If termination condition T(b) holds then the number of messages sent will equal the number of messages received. Therefore, by Rules MAC.1 and MAC.2, $\sum_{j=0}^{N-1} in_j = \sum_{j=0}^{N-1} out_j$. Rules MAC.4, MAC.5, MAC.7, and MAC.8 insure that at the completion of a polling wave $tk(i).IN_2 = \sum_{j=0}^{N-1} in_j$, and $tk(i).OUT_2 = \sum_{j=0}^{N-1} out_j$, where in_j and out_j indicate the values of these variables when the polling events occurred. Successive polling waves will be instigated until a valid polling wave occurs (Rule MAC.7). Therefore, eventually a polling wave, $PW(i)$, will occur such that $tk(i).IN_2 = \sum_{j=0}^{N-1} in_j = tk(i).OUT_2 = \sum_{j=0}^{N-1} out_j$. It is possible that $tk(i).IN_2 = tk(i).IN_1 = tk(i).OUT_2 = tk(i).OUT_1$. If so then $PW(i)$ is a valid wave, and termination is detected. If not then Rules MAC.6 and MAC.7 specify the instigation of $PW(i+1)$. Rule MAC.7 requires that $tk(i+1).IN_1 = tk(i).IN_2$, and $tk(i+1).OUT_1 = tk(i).OUT_2$. Termination conditions T(a) and T(b) hold for $PW(i)$. Therefore, the message counts accumulated in out_j and in_j will not change,

and $tk(i+1).IN_1 = tk(i+1).IN_2 = tk(i+1).OUT_1 = tk(i+1).OUT_2$. By Rule MAC.6, $PW(i+1)$ will be a valid wave, and termination will be detected. ■

Mattern's second algorithm is also based on counting the number of messages sent and received. In this protocol, which he calls the vector count method, the number of messages in and out is tracked per process. This extra information is used to detect termination with fewer messages.

Each process, p_j , maintains a vector of counts, $count_j[0], \dots, count_j[N-1]$. The value of $count_j[i], i \neq j$, is equal to the number of messages sent from p_j to p_i since the last visit of the control wave. The value of $count_j[j]$ is decreased by one each time a message is received. The control wave consists of a control message vector which circulates in a virtual ring imposed on the processors. The control message vector, $tk(i).count$ carries information about the number of messages sent and received, around the ring of processes. When the control vector reaches p_i , $count_i$ is set to $count_i + tk(i).count$. If the result of the summation is vector of all zeros, termination is declared. If there is a non-zero element of $count_i$, then there is some message en route to p_i . There is no point in the wave continuing until this message arrives so the control vector stops at p_i until p_i becomes idle, and p_i has received the number of messages equal to the value of $count_i[i]$. When this occurs, the vector is checked again to determine whether every element is zero. If not $tk(i).count$ is set to $count_i$, $count_i$ is set to zero, and the control message is propagated to $p_{i+1 \bmod N}$. If every element of the vector is zero then all the messages sent have been received, and termination has occurred.

Fully Asynchronous System - Mattern's Vector Count Protocol

MAV.1 A send s such that $\sigma(s) = p_j$ and $\rho(s) = p_k$ sets $count_j[k] = count_j[k] + 1$.

MAV.2 A receive $\eta(s)$ such that $\rho(s) = p_j$ sets $count_j[j] = count_j[j] - 1$.

MAV.3 The occurrence of $w_j(i)$ implies

$$\left[\begin{array}{l} Idle(c_j(i)) \wedge \\ c_j(i) \mapsto w_j(i) \wedge \\ count_j[j] = 0 \end{array} \right] \vee \left[\begin{array}{l} \neg Idle(c_j(i)) \wedge \\ c_j(i) \rightarrow ie_j(k) \mapsto w_j(i) \wedge \\ c_j(i) \rightarrow e'_j \rightarrow ie_j(k) \text{ implies } e'_j \text{ is a send or receive } \wedge \\ count_j[j] = 0. \end{array} \right]$$

MAV.4 When $w_j(i)$ occurs $tk(i).count[k] = count_j[k] \wedge count_j[k] = 0$, for $k = 0, \dots, N-1$.

MAV.5 When $c_j(i)$ occurs $count_j[k] = count_j[k] + tk(i).count[k]$, for $k = 0, \dots, N-1$.

MAV.6 $PW(i)$ is valid if $count_j[k] = 0$ at $w_j(i)$ for some $p_j \in \Pi$, and for $k = 0, \dots, N-1 \wedge i > 1$.

As in Misra's algorithm the final wave may not be complete, i.e. some p_j such that $j < N-1$ might detect termination. While it is possible to prove that this protocol meets our termination conditions, this characteristic makes it difficult notationally. For ease of exposition, it makes sense to slightly redefine the polling wave.

Let termination be detected on the i^{th} visit of the control message to p_j . This means that all elements of $tk(i).count$ were not zero at $w_j(1), w_j(2), \dots, w_j(i-1)$, and that $w_j(2), \dots, w_j(i-1)$ each mark the end of a failed control wave. For purposes of proof we can neglect $w_0(1), w_1(1), \dots, w_{j-1}(1)$ and designate $w_j(1)$ as the beginning of $PW(1)$ and transform the process numbers accordingly. This is possible because a complete circuit of the processes is required for initialization. So, any execution of the algorithm will guarantee that the process which detects termination is visited at least twice by the control message. Therefore, in the following discussion, a valid control wave $PW(i)$ will imply that all elements of $count_{N-1}$ equal zero at $w_{N-1}(i)$, and some element of $count_k$ at $w_k(i)$, $0 \leq k < N-1$ was non-zero, with the understanding that p_{N-1} actually designates p_j , where p_j is the process which detects termination.

Lemma 16 For any $w_j(i)$ event as it is specified in Mattern's Vector Count Protocol

$Idle(w_j(i))$.

Proof: See Lemma 13. ■

Lemma 17 *If polling wave, $PW(i)$, as specified in Mattern's Vector Count Protocol is valid, Termination Condition $T(b)$ is satisfied.*

Proof: We will show that termination condition $T(b)$ holds by assuming the opposite. Let $PW(i)$ be a valid polling wave, and $count_{N-1}[k] = 0, 0 \leq k \leq N - 1$. Assume there exists $\eta(s)$, such that $\rho(s) = p_k$, and $w_k(i) \rightarrow \eta(s)$, and that $\eta(s)$ is the earliest such receive. By Lemma 16, $s \rightarrow w_m(i)$. Such a send will increment $count_m[k]$ by 1. In order for $PW(i)$ to be a valid wave, $count_{N-1}[k] = 0$. This can only happen if at $w_k(i)$, $count_k[k] = -(\sum_{j \neq k} count_j[k])$. So if $count_m[k]$ is incremented by 1, $count_k[k]$ must be decremented by the receipt of a message at p_k before $w_k(i)$. By our original assumption $w_k(i) \rightarrow \eta(s)$. Hence, this message receipt cannot cause $count_k[k]$ to be decremented as necessary. Therefore, there must exist a receipt $\eta(s^*)$ such that $\rho(s^*) = p_k$, and $\eta(s^*) \rightarrow w_k(i)$. The corresponding send must originate at some process after $PW(i)$ so that $\sigma(s^*) = p_g$, and $w_g(i) \rightarrow s^*$. Otherwise, the send will be counted in the p_g count vector, throwing the balance off again. p_g is idle at $w_g(i)$, so this is not possible. ■

Theorem 5 *The completion of a valid wave in the Mattern's Vector Count Protocol satisfies Termination Conditions $T(a)$ and $T(b)$.*

Proof: Follows directly from Lemmas 16 and 17 ■

The proof that Mattern's Vector Count Protocol will detect termination when it occurs closely follows the proof of Mattern's Four Counter Protocol, so it will not be presented here.

We have gone into great detail in presenting the existing protocols in our causal framework. We have done this for two reasons. First, we wanted to illustrate how the polling

wave model and causal reasoning can be used to develop a better understanding of the termination detection problem and the protocols designed to solve this problem. Second, we felt that it was important to show the impact of various system environments on the solution requirements and protocol designs.

Our discussion of these various termination detection protocols amply demonstrates that the concept of causality as it is defined by \rightarrow is basic to understanding the termination problem and its solutions. This is the case because time and the temporal ordering of events is an integral part of the problem. Note both Francez and Dijkstra's specifications for termination of a distributed computation involve an implicit notion of time in the requirement that processors be idle simultaneously.

Only Rana's protocol is able to satisfy this requirement directly. It accomplishes this by requiring perfectly synchronized physical clocks. Given this system characteristic, this protocol can establish that each process is idle at the same time as the process with latest active timestamp.

The other protocols we presented are designed for systems in which the processors do not have access to global time, and as we pointed out in the beginning of our discussion of termination detection, verifying simultaneity in the absence of global time is impossible. So instead, each of the algorithms designed for asynchronous system utilizes a series of waves and the causal relationships between the waves as a substitute for real time. These protocols report termination when all processes are idle for some time interval, the beginning and end of which is defined only relative to the events of two successive waves. So the required condition that all process are idle simultaneously can be inferred in spite of the fact that no specific physical time can be identified. The common element in all of these polling algorithms is the use of *causal intervals*, time intervals which are defined in terms of event orderings.

In the following section we show how vector clocks can be readily substituted for real time in Rana's protocol. Thus the need for synchronized clocks can be eliminated without having to resort to the causal intervals used in the other polling protocols.

2.4 Causal Termination Detection Protocol

Rana's algorithm with its use of physical time illustrates the usefulness of global time in designing a termination detection algorithm. The availability of globally significant time makes it simple to establish that processes are idle at the same time and eliminates the need to create these causally defined intervals of time. However, global time is a very expensive requirement to impose on a system. Instead of using real time or causal intervals, we use logical clocks and timestamps in ways analogous to physical timestamps to design termination detection algorithms.

We pointed out in the introduction that vector clocks allow the complete partial order to be deduced. For this reason we will use vector clocks rather than Lamport's logical time in our protocol. Before we proceed we need to modify the definition of vector time so that it can be used with synchronous communication.

2.4.1 Vector Clocks for Synchronous Communication

Causality, Lamport's logical clocks, and vector time were defined primarily for systems which are based on asynchronous communication. The use of synchronous communication changes these relationships somewhat as we saw when we defined \rightarrow for a synchronous send and receive. No longer does a send happen before a receive. The vector times of the send and receive should reflect that by being equal. Accordingly, the previous set of rules for calculating vector clock values are modified to insure that timestamp of a send is set equal to that of the receive.

1. When event e'_i occurs in p_i , $V_i^i = V_i^i + 1$. The clock value of e'_i is $V_i(e'_i)$.

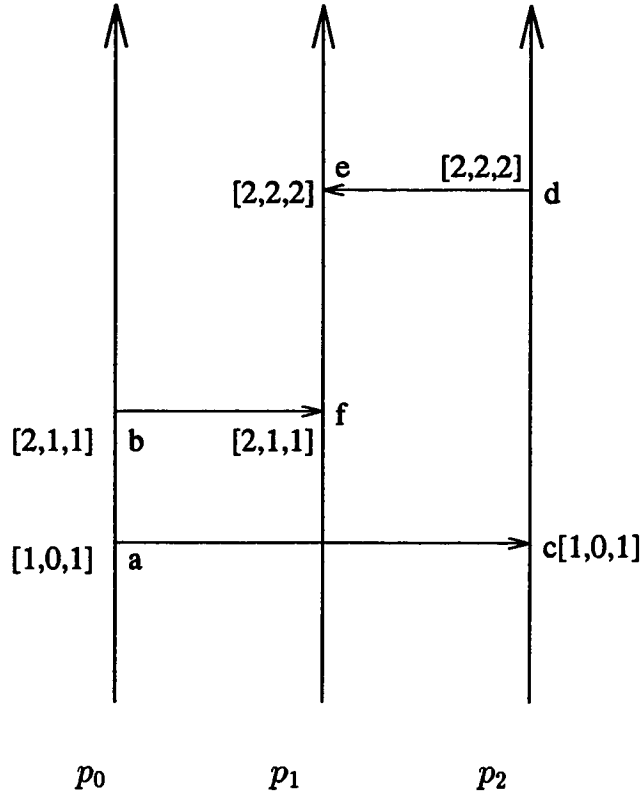


Figure 2.3: Vector Time - Synchronous Communication

2. If e'_i is a send in p_i , and e'_j a receive in p_j , then the clock value of e'_j is updated to reflect the clock value of e'_i so that $V_j(e'_j)$ is assigned $V_j = \text{sup}(V_j, V_i)$, where $\text{sup}(V_i, V_j) = \max(V_i^k, V_j^k)$, for $0 \leq k \leq N - 1$.
3. After V_j is calculated, V_i must be set to the value of V_j .

Figure 2.3 illustrates how vector time is calculated with synchronous communication. Note that in this environment event c happens before event b . This is reflected in the clock values of c and b , $V(c) < V(b)$.

The following rules summarize the relationship between vector time, causality, and synchronous communication:

Rules for Causality and Vector Time under Synchronous Communication

- For any events e'_i and e'_j , $e'_i \rightarrow e'_j$ iff $V(e'_i) < V(e'_j)$.
 - For send and receive events s and $\eta(s)$, $V(s) = V(\eta(s))$.
-

Asynchronous Systems - Synchronous Communication

We will now show some examples of how logical time as maintained by vector clocks can be used instead of physical time to bring an illusion of synchrony to an asynchronous system. First we will show how this principle can be applied to an asynchronous system which uses synchronous communication. The causal protocol we derive is modeled after Rana's algorithm which is applicable to synchronous systems with synchronous communication. Rana's protocol used physical timestamps to determine the latest time a process was active. In an asynchronous system where every send and receive has a vector timestamp, these timestamps can be used to determine which processes are active latest, at least within a group that are causally related.

In a synchronous system there can only be one latest physical time because the times of the events are totally ordered. In a causally synchronous system timestamped events are only partially ordered. So, there could be several events with concurrent timestamps, from which we could not distinguish a single latest event. To accommodate this difference, the circulating token we will use to detect termination will contain a set of vector clock values from several processes rather than a single process timestamp. This set will have the property that every vector timestamp in the set is concurrent to every other clock vector in the set. In the discussion and proof of our protocol, we will show that an appropriately constructed set of vector times can be used to determine which processes were last active

in a manner similar to Rana's use of a physical timestamp.

Before we describe the algorithm and the means for constructing this set of vector times, we need to state some assumptions and definitions. In this protocol, the vector time of processes is updated only by basic communication. Messages of the termination detection algorithm are not counted in the calculation of the vector time in the underlying computation. All communication is synchronous. Any token specified in the protocol circulates in a virtual ring imposed on the processes, so that a token propagated by p_i is sent to $p_{i+1 \bmod N}$. The vector time of process p_i is denoted by V_i , and the timestamp of an event e'_i in p_i is given by $V_i(e'_i)$.

In this protocol, as in those we have previously discussed, a token is used to create a control wave for the purposes of detecting termination. Actually, a series of tokens may be generated and circulate in the virtual ring. The implementation rules insure that at least one token completes the circuit and such a completion indicates termination.

A process may either *generate* a token or *propagate* a token. When a token is generated, a new token is created with a new timestamp set. A process which propagates a token makes no changes to the content of a token. A process propagates a token if the token arrives when p_i is idle, and $V_i \leq V_j$, for some V_j in the timestamp set of the incoming token.

A token may be generated under two circumstances. In the first case, a token is generated when a process becomes idle after some period of activity. The timestamp set in the token generated by a process p_i , is a singleton containing p_i 's current vector time, V_i . In the second case, a process generates a new token if it is idle when a token created by another process arrives, and the timestamp in this arriving token must be modified. The timestamp of a token arriving at an idle process must be modified if the vector time of the idle process is either concurrent, but not equal to, every clock value in the timestamp set of the arriving token, or greater than any one of the clock values in the set.

In the case where the value of V_i is concurrent to each member of the set in the incoming token, the vector V_i is added to the set to create the timestamp for the new token. If the value of V_i is greater than any vector in the incoming token, that vector is deleted from the set, and V_i is added to the set. The function *GenerateToken* is used to perform the construction of the timestamp for the token.

Function *GenerateToken*($V_i, timestamp$)

If $\exists V_j \in timestamp$ such that $V_i > V_j$ then

$\{ \forall V_j \in timestamp \text{ such that } V_i > V_j$

$DeleteSet = DeleteSet \cup V_j$

$return = V_i \cup timestamp - DeleteSet$

}

otherwise,

$return = V_i \cup timestamp.$

As in Rana's protocol, each token is associated with its creator by an identification stamp, and because a process may generate several tokens that may exist concurrently, an additional index number must be assigned to uniquely identify each token. Therefore, the n^{th} token generated by p_i will be signified by $tk(i, n)$, and its timestamp will be $tk(i, n).ts$. In this algorithm there is only one complete wave. In fact, completion of a wave indicates termination. For this reason events w and c are redefined. The definitions are identical to that used in the description of Rana's algorithm. The waves are no longer identified by the number completed, but rather by the originating process and token index.

Causal Termination Detection - Synchronous Communication

CTS.1 $w_j(i, m), i \neq j$ occurs iff

$$\begin{aligned} & Idle(c_j(i, m)) \wedge \\ & \exists V_i \in tk(i, m).ts \text{ such that } V_j \leq V_i \wedge \\ & c_j(i, m) \mapsto w_j(i, m). \end{aligned}$$

CTS.2 $w_j(j, m)$ occurs iff

$$\begin{aligned} & \exists ie_j(k) \text{ such that } ie_j(k) \mapsto w_j(j, m) \vee \\ & \left[\begin{array}{l} \exists c_j(i, l) \text{ such that } Idle(c_j(i, l)) \wedge c_j(i, l) \mapsto w_j(j, m) \wedge \\ \quad \nexists V_p \in tk(i, l).ts \text{ such that } V_j \leq V_p. \end{array} \right] \end{aligned}$$

CTS.3 The occurrence of $w_j(j, m)$ implies $tk(j, m).ts = GenerateToken(V_j, timestamp)$

where $timestamp = \emptyset$ when $ie_j(k) \mapsto w_j(j, m)$, and $timestamp = tk(i, l).ts$ when $\exists c_j(i, l)$, such that $c_j(i, l) \mapsto w_j(j, m)$.

CTS4. Polling wave $PW(j, m)$ is valid if $c_j(j, m)$ has occurred for some $p_j \in \Pi$.

The first step in arguing the correctness of this protocol is to show that the set of timestamps in the token has the properties necessary to serve as a surrogate for real time in detecting termination.

Lemma 18 *If termination of the underlying computation occurs then there exist j, n such that $PW(j, n)$ will be a complete wave.*

Proof: Let $T = \{p_i \mid \exists k : V_i < V_k\}$. Each member of this set, T , has an earlier timestamp than at least one other process. Let $S = \Pi - T$. $p_i, p_k \in S$ implies $V_i \parallel V_k$. Let p_j be a

process in S . If the n^{th} token originating at p_j has a timestamp equal to $\{V_i \mid p_i \in S\}$, this token will make a complete circuit of all the processors and produce a valid wave, $PW(j, n)$.

Rule CTS.2 of the protocol requires that each process $p_i \in S$ produce a token when p_i becomes idle. Rule CTS.3 and the definition of *GenerateToken* insures that the timestamp of the token produced when p_i became idle equals V_i . Because V_i is concurrent to V_k for every $p_k \in S$, and because of Rule CTS.3 governing token generation, the value of V_i for all $p_i \in S$ will reach p_j in a token generated by some element of S . When this occurs, p_j will originate a token such that $tk(j, n).ts = \{V_i \mid p_i \in S\}$. For every $p_k \in \Pi$, there exists $V_i \in tk(j, n).ts$ such that $V_i \geq V_k$, therefore, by Rule CTS.1, $w_k(j, n)$ will occur for all p_k . Rule CTS.4 specifies that in these circumstances $PW(j, n)$ will be a valid wave. ■

Lemma 19 *Let $tk(j, n).ts$ equal the timestamp of the token propagated or generated at $w_i(j, n)$. If there exists $V_k \in tk(j, n).ts$ such that $V_k \geq V_i(e'_i)$ then $e'_i \rightarrow w_i(j, n)$.*

Proof: Let $V_k \in tk(j, n).ts$, and $V_k \geq V_i(e)$. Also assume that $e'_i \not\rightarrow w_i(j, n)$. Because e'_i and $w_i(j, n)$ are in the same process, e'_i can't be concurrent to $w_i(j, n)$. That leaves the possibility that $w_i(j, n) \rightarrow e$. This would imply that $V_i(e'_i) > V_i(w_i(j, n))$, and $V_i^i(e'_i) > V_i^i(w_i(j, n))$. However, the rules for calculating vector time require that $V_i^i \geq V_j^i$ for all $p_j \neq p_i$. Therefore, $V_i^i(e'_i) > V_m^i$ for all $V_m \in tk(j, n).ts$, contradicting our original assumption. ■

Lemma 20 *For any $w_j(i, n)$ event as it is specified in the Causal Termination Detection - Synchronous Communication Protocol, $Idle(w_j(i, n))$.*

Proof: Assume that there exists $w_j(i, n)$ such that $\neg Idle(w_j(i, n))$. If this were the case, then there exists $\eta(s)$ such that $\rho(s) = p_j$, and $\eta(s) \rightarrow w_j(i, n)$. In addition there would not exist an idle event $ie_j(k)$ such that $\eta(s) \rightarrow ie_j(k) \rightarrow w_j(i, n)$. This directly contradicts Rule CTS.2 which requires that there exist a $ie_j(k)$ such that $ie_j(k) \mapsto w_j(i, n)$. $\neg Idle(w_j(i, n))$ also contradicts Rule CTS.2. If $Idle(c_j(i, n))$ then there exists $ie_j(k)$, such

that $ie_j(k) \rightarrow c_j(i, n)$ without an intervening receive or send, and $c_j(i, n) \rightarrow w_j(i, n)$ without an intervening event. Hence, $ie_j(k) \rightarrow w_j(i, n)$, and there is no $\eta(s)$ such that $ie_j(k) \rightarrow \eta(s) \rightarrow w_j(i, n)$. ■

Lemma 21 *If polling wave, $PW(i, n)$, as specified in the Causal Termination Detection - Synchronous Communication Protocol, is valid then Termination Condition $T(b)$ is true.*

Proof: If $T(b)$ does not hold then there exists $\eta(s)$ such that $w_{\rho(s)}(i, n) \rightarrow \eta(s)$. There are two possible cases, either $w_{\rho(s)}(i, n) \rightarrow s \rightarrow w_{\sigma(s)}(i, n)$, or $w_{\sigma(s)}(i, n) \rightarrow s$. By Lemma 20, $p_{\sigma(s)}$ is idle at $w_{\sigma(s)}(i, n)$ so it is not possible that $w_{\sigma(s)}(i, n) \rightarrow s$. The other case is that $s \rightarrow w_{\sigma(s)}(i, n)$ which implies $V_{\sigma(s)}(s) \leq V_{\sigma(s)}(w_{\sigma(s)}(i, n))$. Because communication is synchronous, $V_{\sigma(s)}(s) = V_{\rho(s)}(\eta(s))$. $PW(i, n)$ is a valid wave so by Rule CTS.1, there exists $V_l \in tk(i, n).ts$ such that $V_l \geq (V_{\sigma(s)}(w_{\sigma(s)}(i, n)) \geq V_{\sigma(s)}(s))$. So there exists $V_l \in tk(i, n).ts$ such that $V_l \geq V_{\rho(s)}(\eta(s))$. By Lemma 19, this implies that $\eta(s) \rightarrow w_{\rho(s)}(i, n)$, contradicting our original assumptions. ■

Theorem 6 *The completion of a valid wave in the Causal Termination Detection - Synchronous Communication Protocol satisfies Termination Conditions $T(a)$ and $T(b)$.*

Proof: Follows directly from Lemmas 20 and 21. ■

2.4.2 Asynchronous System - Asynchronous Communication

We have shown how vector time can be utilized in an environment where communication is synchronous. The next logical step is to examine how vector time can be substituted for real time to detect termination in a system with asynchronous communication. We will use an approach similar to that we used to develop the synchronous communication algorithm. That is solving the problem as if synchronized clocks were available, and then modifying it to use vector time instead.

There are no published protocols for detecting termination given a system of synchronized clocks and asynchronous communication, so we will present two we have developed.

Rana's protocol will not work correctly if communication is asynchronous because there is no guarantee that messages sent before or during a final wave will arrive during the final wave. As we have shown in our correctness arguments, without this guarantee a wave might complete showing all processes to be idle, yet a message could be in transit. An obvious way to modify Rana's algorithm is to keep track of all the messages which have been sent and not declare termination until they have been received. This can be done by associating a physical timestamp with each message and using the token to carry this information to the processes so that the messages sent can be matched to those received.

This protocol operates as follows. Each process maintains two sets of timestamps. The set R_i contains the timestamps of messages received by p_i . The set S_i contains the timestamps of messages sent by p_i . Any process, p_i , upon becoming idle generates a token composed of S_i , the current timestamp, $T(p_i)$, and $id(p_i)$. This token is sent to $p_{i+1 \bmod N}$.

Any p_i which receives a token checks to see if there is a timestamp in S_{tk} , the send set carried by the token, that matches a timestamp in R_i . If any are found, they are deleted from the two sets. An active process, p_i , which receives a token removes any matches from S_{tk} and R_i as described above. It then saves the remaining timestamps in S_{tk} by incorporating them into its own set of timestamps, so that $S_i = S_i \cup S_{tk}$. The incoming token is then discarded. An idle process, p_i , which receives a token also removes any matches. p_i then compares its own timestamp $T(p_i)$ to $T(tk)$ to determine what action to take. If $T(p_i) > T(tk)$, p_i proceeds as if it were active, saving S_{tk} and discarding the token. If $T(p_i) < T(tk)$, the send set of p_i is added to the token's send set, and the token is propagated to the next process. An idle process which receives its own token back checks whether $S_{tk} = \{\}$. If this is the case then the computation is terminated. If not, then S_{tk} is added to S_i , and the token is discarded.

Clearly this protocol could be modified to use vector timestamps instead of physical

timestamps in the S and R sets. The token timestamp would be maintained exactly as it was in the *Causal Termination Detection* protocol for synchronous communication we presented. Instead of maintaining lists of physical timestamps, vector timestamps would be used.

Causal Termination Detection - Asynchronous Communication (Protocol A)

CTAA.1 A send s such that $\sigma(s) = p_i$, causes V_i to be updated according to the rules governing vector clocks and $S_i = S_i \cup V_i(s)$.

CTAA.2 A receive $\eta(s)$ such that $\rho(s) = p_j$, and $\sigma(s) = p_i$, updates V_j according to the rules governing vector clocks, and $R_j = R_j \cup V_i(s)$.

CTAA.3 The occurrence of $c_j(i, m)$ implies

$$\begin{aligned} \forall ts, ts' : ts \in R_j \wedge ts' \in S_{tk} \wedge ts = ts', R_j = R_j - ts, S_{tk} = S_{tk} - ts' \wedge \\ S_j = S_j \cup S_{tk}. \end{aligned}$$

CTAA.4 $w_j(i, m), i \neq j$ occurs iff

$$\begin{aligned} & Idle(c_j(i, m)) \wedge \\ & \exists V_i \in tk(i, m).ts \text{ such that } V_j \leq V_i \wedge \\ & c_j(i, m) \mapsto w_j(i, m). \end{aligned}$$

CTAA.5 $w_j(j, m)$ occurs iff

$$\begin{aligned} & \exists ie_j(k) \text{ such that } ie_j(k) \mapsto w_j(j, m) \vee \\ & \left[\begin{array}{l} \exists c_j(i, l) \text{ such that } Idle(c_j(i, l)) \wedge \\ c_j(i, l) \mapsto w_j(j, m) \wedge \\ \nexists V_p \in tk(i, l).ts \text{ such that } V_j \leq V_p. \end{array} \right] \end{aligned}$$

CTAA.6 The occurrence of $w_j(j, m)$ implies $tk(j, m).ts = \text{GenerateToken}(V_j, \text{timestamp})$ where $\text{timestamp} = \emptyset$ when $ie_j(k) \mapsto w_j(j, m)$, and $\text{timestamp} = tk(i, l).ts$ when $\exists c_j(i, l)$, such that $c_j(i, l) \rightarrow w_j(j, m)$.

CTAA.7 Occurrence of $w_j(j, m)$ implies $S_{tk} = S_j$.

CTAA.8 Occurrence of $w_j(i, m)$ implies $S_{tk} = S_{tk} \cup S_j$.

CTAA.9 A polling wave, $PW(i, m)$, is valid if $c_j(j, m)$ has occurred for some $p_j \in \Pi$ and $S_{tk} = \{\}$.

CTAA.10 If $c_j(j, m)$ occurs, and $S_{tk} \neq \{\}$, the token is discarded.

Unfortunately, neither the synchronous protocol we informally described or the causal protocol specified above is very efficient. The token in both cases may end up carrying an unrealistic amount of information with it as it transits the cycle of processors. It is not necessary to maintain a record of messages sent and received in such detail to solve this problem. The reader will recall that *Mattern's Vector Count Protocol* was able to detect termination by tracking the messages received per process. Global clocks can be used to reduce the amount of detail required even further to a simple balancing of messages in and messages out. Mattern's Four Counter protocol used this technique, but it required two successive waves to prevent counting of messages from the future. The availability of global time can be used to design a protocol that counts total messages in and out and only requires a single wave.

Such an algorithm is very similar to Rana's. Additional variables are required to accumulate the message counts. Each process, p_i , maintains counters in_i and out_i to count the messages sent and received. The token carries two counters, IN and OUT to accumulate the total messages sent and received.

Synchronous Termination Detection Protocol - Asynchronous Communication

STA.1 A send s such that $\sigma(s) = p_i$, sets $in_i = in_i + 1$.

STA.2 A receive $\eta(s)$ such that $\rho(s) = p_j$, sets $out_j = out_j + 1$.

STA.3 $w_j(i, m), i \neq j$ occurs iff

$$\begin{aligned} &Idle(c_j(i, m)) \wedge \\ &T(ie_j(k)) < T(tk_i) \wedge \\ &c_j(i, m) \mapsto w_j(i, m) \wedge \\ &\exists ie_j(m) \text{ such that } ie_j(k) \rightarrow ie_j(m) \rightarrow c_j(i, m). \end{aligned}$$

STA.4 $w_j(j, m)$ occurs iff $\exists ie_j(k)$ such that $ie_j(k) \mapsto w_j(j, m)$.

STA.5 Occurrence of $w_j(i, m), i \neq j$ implies $IN = IN + in_j$, and $OUT = OUT + out_j$.

STA.6 The occurrence of $w_j(j, m)$ implies $T(tk_j) = T(ie_j(k))$ where $ie_j(k) \mapsto w_j(j, m)$.

STA.7 Occurrence of $w_j(j, m)$ implies $IN = in_j$, and $OUT = out_j$.

STA.8 A polling wave, $PW(j, m)$, is valid if $c_j(j, m)$ has occurred for some $p_j \in \Pi$, and

$$OUT = IN.$$

STA.9 Initialization: $in_i = out_i = 0$, for all $p_i \in \Pi$.

By using our previously defined method for maintaining a vector timestamped token this synchronous protocol can be easily modified to work in a totally asynchronous environment. This modified protocol is defined in the following section. The reader will note the similarity to our causal protocol for synchronous communication.

Causal Termination Detection - Asynchronous Communication Protocol(Protocol B)

CTAB.1 A send s such that $\sigma(s) = p_i$, and $\rho(s) = p_j$ cause $in_i = in_i + 1$, and $out_j = out_j + 1$.

CTAB.4 $w_j(i, m), i \neq j$ occurs iff

$$\begin{aligned} & Idle(c_j(i, m)) \wedge \\ & \exists V_i \in tk(i, m).ts \text{ such that } V_j \leq V_i \wedge \\ & c_j(i, m) \mapsto w_j(i, m). \end{aligned}$$

CTAB.5 $w_j(j, m)$ occurs iff

$$\begin{aligned} & \exists ie_j(k) \text{ such that } ie_j(k) \mapsto w_j(j, m) \vee \\ & \left[\begin{array}{l} \exists c_j(i, l) \text{ such that } Idle(c_j(i, l)) \wedge \\ c_j(i, l) \mapsto w_j(j, m) \wedge \nexists V_p \in tk(i, l).ts \text{ such that } V_j \leq V_p. \end{array} \right] \end{aligned}$$

CTAB.6 The occurrence of $w_j(j, m)$ implies $tk(j, m).ts = GenerateToken(V_j, timestamp)$ where $timestamp = \emptyset$ when $ie_j(k) \mapsto w_j(j, m)$, and $timestamp = tk(i, l).ts$ when $\exists c_j(i, l)$, such that $c_j(i, l) \mapsto w_j(j, m)$.

CTAB.4 Occurrence of $w_j(j, m)$ implies $IN = in_j$, and $OUT = out_j$.

CTAB.5 The occurrence of $w_j(i, m)$ implies $IN = IN + in_j$, and $OUT = OUT + out_j$.

CTAB.6 A polling wave, $PW(j, m)$ is valid if $c_j(j, m)$ has occurred for some $p_j \in \Pi$, and $OUT = IN$.

CTAB.7 If when $c_j(j, m)$ occurs, and $OUT \neq IN$, then discard token.

CTAB.8 Initialization: $in_i = out_i = 0$.

Lemma 22 *For any $w_j(i, m)$ event as it is specified in the Causal Termination Detection - Asynchronous Communication (Protocol B), $Idle(w_j(i, m))$.*

Proof: See Lemma 20.

Lemma 23 *If polling wave, $PW(i, m)$, as specified in the Causal Termination Detection - Asynchronous Communication (Protocol B) is valid then Termination Condition $T(b)$ is true.*

Proof: If Termination Condition $T(b)$ does not hold there must exist $\eta(s)$ such that $w_{\rho(s)}(i, m) \rightarrow \eta(s)$. Let $\eta(s)$ be the earliest such receive. By Lemma 22, the corresponding send must have originated before the wave. Therefore, $s \rightarrow w_{\sigma(s)}(i, m)$. This would imply $IN \leq OUT - 1$ at $c_i(i, m)$ unless there is a balancing receive, $\eta(s')$ such that $\eta(s') \rightarrow w_{\rho(s')}(i, m)$, and $w_{\sigma(s')}(i, m) \rightarrow s'$ (Rules CTAB.1, CTAB.4, and CTAB.5). Because $PW(i, m)$ is a valid wave, $IN = OUT$ at $c_i(i, m)$ (Rule CTAB.6), and such a balancing receive must exist. $s' \rightarrow w_{\rho(s')}(i, m)$ implies $V_{\sigma(s')}(s') < V_{\rho(s')}(\eta(s')) < V_{\rho(s')}(w_{\rho(s')}(i, m))$. Rules CTAB.2 and CTAB.3 specify that $V_{\rho(s')}(w_{\rho(s')}(i, m)) \in tk(i, m).ts$, or there exists $V_k \in tk(i, m).ts$ such that $V_k > V_{\rho(s')}(w_{\rho(s')}(i, m))$. In either case there exists a vector in $tk(i, m).ts$ of greater value than $V_{\sigma(s')}(s')$. By Lemma 19, that implies $s' \rightarrow w_{\sigma(s')}(i, m)$, contradicting the premise that $\eta(s')$ is a balancing receive. ■

Theorem 7 *The Causal Termination Detection Protocol for Asynchronous Communication (Protocol B) satisfies Termination Conditions $T(a)$ and $T(b)$.*

Proof: Follows directly from Lemmas 22 and 23. ■

The two causal termination protocols we have presented illustrate the utility of vector time. They also demonstrate how vector time can be used to stand in for real time in

a straightforward way. Vector time does not eliminate the need to poll every process at least once after the process becomes idle, but this is a characteristic of every termination detection algorithm. Vector time does enable us to design algorithms which only require one final wave. This claim may only be made for Rana's synchronous protocol and Mattern's Vector count protocol. The remaining protocols require that each process be polled twice after they become idle.

In Chapter 3 we will apply the methodology of polling waves, causal correctness conditions, causal protocol specifications, and vector time to the problem of detection and resolution of distributed deadlock. Deadlock is similar to termination in the sense that every process in a deadlocked set is idle. However, termination is a stable property of a system. Once a computation has terminated it remains terminated. The purpose of detecting deadlock is to resolve the deadlock so that computation may proceed, therefore, deadlock is not a stable property when resolution is instigated. The dynamic nature of deadlock detection and resolution makes it a difficult problem to solve. We will show how our causal methodology with its dependence on local state leads to simple and demonstrably correct solutions.

Chapter 3

Distributed Deadlock Detection and Resolution

3.1 Deadlock Detection - System Model

A problem which has been the subject of extensive research is deadlock detection in a distributed system. Two categories of deadlock, communication and resource, can arise in a distributed system.

Communication deadlock occurs when each process in a set of processes is blocked, waiting for a message from some other process in the set. Resource deadlock arises in distributed databases when each process in a set of processes cannot proceed because it is waiting for another process in the set to release a resource. We will limit our discussion to resource deadlock in this section.

A database system is comprised of a static set of d non-terminating *data manager* processes $\mathcal{D} = \{D_1, \dots, D_d\}$, a set of *data resources* \mathcal{R} , a static set of t non-terminating *transaction manager* processes $\mathcal{TM} = \{TM_1, \dots, TM_t\}$, and a set of *transaction* processes \mathcal{T} . Data manager $D_i \in \mathcal{D}$ will be bound to a single node of the network, and it will control access to R_i (some part of the database) which is assumed to reside in some storage device

physically located at that node. Similarly, transaction manager $TM_i \in \mathcal{TM}$ also executes at a single node of the network, and it will control a single transaction process. Transaction $T_i \in \mathcal{T}$ will be created at some node of the network and is controlled by the transaction manager associated with that node.

Before a transaction T_i can access a data resource, it must receive access permission from the data manager responsible for the resource. A transaction does not directly communicate with a data manager to obtain this permission. All communication with a transaction, T_i , is routed through the transaction manager that controls T_i . However, it is cumbersome to continually refer to both the transaction manager and the transaction when describing the operation of the system, therefore, in the remainder of this discussion it should be assumed that any communication ascribed to a transaction is actually performed by a transaction manager. Data managers and transaction managers communicate solely by explicit message passing.

We assume that two-phase locking is used for concurrency control. To that end, a transaction will send a *request* message, through its transaction manager, to lock a data resource. The data manager will reply with a *grant* message if the lock is granted. Otherwise a *hold* message will be sent to the requesting transaction by the data manager to indicate that the data resource is locked by another transaction and that the lock request has been enqueued. A transaction may not proceed until it has acquired locks on all the resources it needs. Once a transaction has all the necessary locks, it can read and write the data resource. When a transaction no longer needs the data resource (after the transaction has committed or aborted its changes) it releases the resource. When the transaction releases the resource, the controlling transaction manager sends a *release* message to the appropriate data manager. Once a transaction releases a lock it may make no further requests.

We identify some specific events in the distributed database system:

- $\text{sendReq}_{T_i \rightarrow D_j}^k$, is the transmission of the k th request message from T_i through TM_i to

D_j .

- $\text{recvReq}_{T_i \rightarrow D_j}^k$ is D_j 's receipt of the k th request message send from T_i through TM_i to D_j .
- $\text{sendGrant}_{D_j \rightarrow T_i}^k$ is the transmission of a grant message from D_j to TM_i in response to the request sent by $\text{sendReq}_{T_i \rightarrow D_j}^k$.
- $\text{recvGrant}_{D_j \rightarrow T_i}^k$ is TM_i 's receipt of the grant message sent by $\text{sendGrant}_{D_j \rightarrow T_i}^k$.
- $\text{sendHold}_{D_j \rightarrow T_i}^k$ is the transmission of a hold message from D_j to TM_i in response to the request sent by $\text{sendReq}_{T_i \rightarrow D_j}^k$.
- $\text{recvHold}_{D_j \rightarrow T_i}^k$ is TM_i 's receipt of the hold message sent by $\text{sendHold}_{D_j \rightarrow T_i}^k$ in response to the request sent by $\text{sendReq}_{T_i \rightarrow D_j}^k$.
- $\text{sendRel}_{T_i \rightarrow D_j}^k$ is the transmission of a release message from T_i through TM_i to D_j , releasing the datum requested by $\text{sendReq}_{T_i \rightarrow D_j}^k$.
- $\text{recvRel}_{T_i \rightarrow D_j}^k$ is D_j 's receipt of the release message sent by $\text{sendRel}_{T_i \rightarrow D_j}^k$.
- e_i' - generic event in D_i or T_i

Figure 3.1 illustrates the interactions between transactions, transaction managers, and data managers in a distributed database system. Deadlock can arise if a cycle of transactions is formed with each member of the cycle waiting for a resource held by some other transaction in the cycle. In the example, T_1 and T_4 are deadlocked because each transaction is waiting for a resource held by the other transaction.

This type of representation is useful for detailed analysis of system and protocol behavior because it explicitly identifies the events that occur in the process execution. The complexity of this representation obscures the wait-for relationships that we want to identify. The

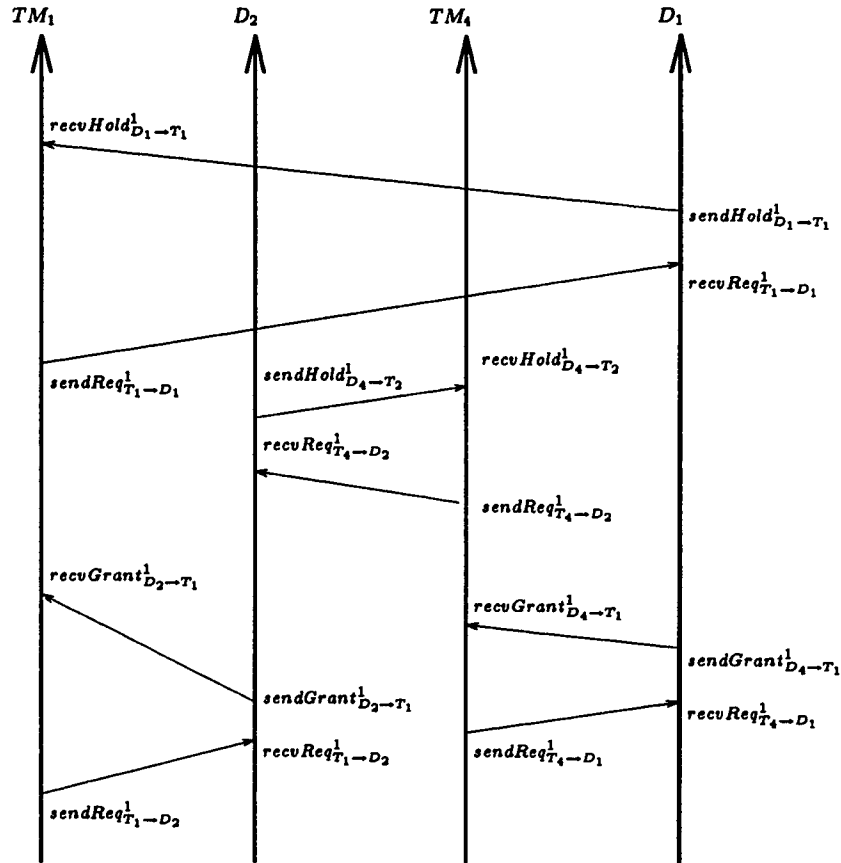


Figure 3.1: Data Manager and Transaction Execution

graphic representation of transaction activity shown in Figure 3.2 is much simpler. This representation treats requests and the corresponding responses as atomic events. Ignoring the details of the message events that occur when requests are made and resources are granted makes it easier to visualize the relationships that develop between transactions and data managers.

In this example there are six transactions. A directed arc between two transactions in the graph indicates that one transaction is waiting for a resource held by another transaction. So, in this example, T_1 waits for resource R_9 held by T_2 , T_2 waits R_1 held by T_3 , and so on. Once this cycle is formed the transactions in the cycle are idle and will remain idle until

some action is taken to resolve the deadlock. The object of resource deadlock detection is to determine that such a cycle exists.

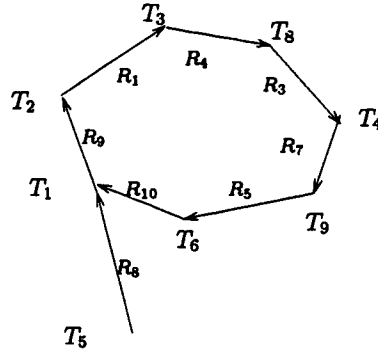


Figure 3.2: Transaction-Wait-For Graph

Conceptually deadlock is very similar to termination. If all the transactions in a system are in a deadlock cycle then they will appear idle just as if they were terminated. A deadlock cycle, however, is usually restricted to a subset of system transactions. A termination protocol will not work properly when this is the case. There is another difference between termination and deadlock. While termination is a permanent state, the deadlocked or “terminated” set will not remain idle permanently if the protocol acts to resolve the deadlock. During resolution one transaction is usually aborted, and the remaining processes may become active again. These differences have made deadlock detection and resolution a difficult problem to solve. Numerous protocols for solving this problem have been presented in the literature. In general they are complex and expensive. More disturbing is that most have been shown to be incorrect. In the following section we will discuss some of these protocols and point out some of the problems.

3.1.1 Previous Research

In a traditional multiprocessing system with shared memory and centralized control, deadlock detection protocols construct and maintain graphs similar to the one shown in Fig-

ure 3.2. These graphs are called transaction-wait-for (TWF) graphs. Each time a request is made which can not be granted an arc is added to the graph. Arcs are deleted from the graph when locks are released, and waiting transactions are granted resources. Each time an arc is added to the graph the detection protocol checks for the creation of a cycle that indicates deadlock.

Early deadlock detection protocols were modeled on this sequential system paradigm [39, 12, 7]. The protocols attempt to construct transaction-wait-for (TWF) graphs distributively, or they impose a centralized control on the system to construct the graphs in a centralized manner. Maintenance of TWF graphs is a viable technique in a system where resource requests and releases are totally ordered, and an accurate global view of the system is facilitated by a shared memory. Accurately maintaining the global view that a TWF graph represents in a distributed system turns out to be extremely complex and costly. The complexity of these protocols also leads to errors.

Isloor and Marsland [39] describe a protocol that maintains a complete TWF graph at every site. This requires the broadcast of each transaction request and release to every site. It also requires careful design to accommodate the indeterminacy of receipt of messages conveying the graph update information. In addition to having high overhead, this protocol has been shown to be incorrect [10].

Menasce and Muntz [12] attempt to reduce the cost of maintaining the TWF graph by constructing condensed, or partial, TWF graphs at each resource controller site. This protocol is also incorrect [10]. We will describe the protocol in some detail to illustrate the difficulty inherent in trying to maintain an up-to-date global view in a distributed system.

The system model Menasce and Muntz use differs slightly from that which we outlined in Section 3.1. In their model each transaction manager resides at a data manager site. This means that a transaction must send a request message only when it needs a resource at a different site from that where it resides.

In their protocol each data manager maintains a condensed TWF graph based on in-

formation generated at its site and information sent to it by other data managers. When a transaction makes a request of a data manager that cannot be met, the data manager adds an arc to its local TWF graph. So, if transaction T_1 makes a request to D_1 , and the resource managed by D_1 is already locked by T_2 , D_1 adds the pair (T_1, T_2) to its TWF graph. If T_1 does not reside at the same site as D_1 , it sends this pair, (T_1, T_2) , to the site where T_1 resides. These pairs are called *blocking pairs*.

When a data manager receives one of these blocking pairs it adds the arc represented by the pair to its own graph. It then examines its local TWF graph. If there is a transaction in the local TWF graph that blocks the second element of the incoming pair, a new pair is generated. This pair will contain the blocked transaction and the blocking transaction. This pair is sent to the site of the blocking transaction. Whenever a blocking pair is added to a local TWF graph the data manager checks for a cycle. The presence of a cycle indicates deadlock has occurred. The following example details the actions of the protocol.

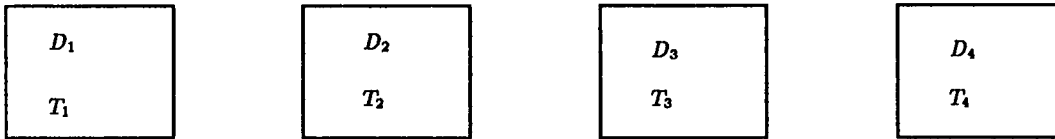


Figure 3.3: Initial Resource Allocation

In Figure 3.3, T_1 has locked R_1 , T_2 has locked R_2 , T_3 has locked R_3 , T_4 has locked R_4 . Each transaction resides at the data site of the resource it has locked. T_1 then makes a request to D_3 for R_3 . D_3 adds the arc (T_1, T_3) to its local TWF graph. D_3 also transmits this pair to D_1 . D_1 will add the pair to its local TWF graph. The situation then appears as shown in Figure 3.4.

T_2 then makes a request for R_4 , and T_4 makes a request for R_1 . D_4 will add (T_2, T_4) to its TWF graph. It will also send (T_2, T_4) to D_2 . When the request from T_4 arrives at D_1 , D_1 will add the pair (T_4, T_1) to its local graph. D_1 will also send (T_4, T_3) to T_4 's site and T_3 's site to indicate that T_4 is waiting transitively on T_3 . Figure 3.5 shows how the TWF

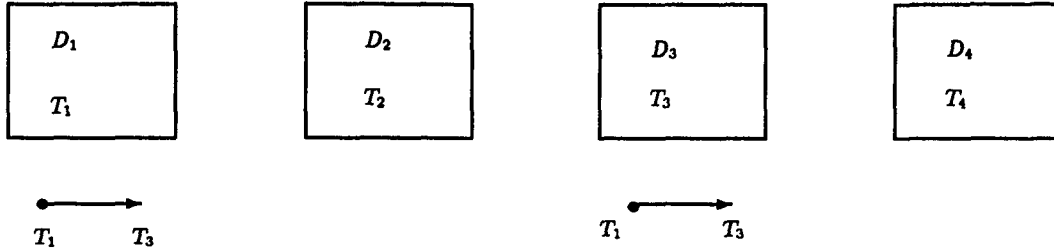


Figure 3.4: Request for R_3

graphs would look at each site when this activity was complete.

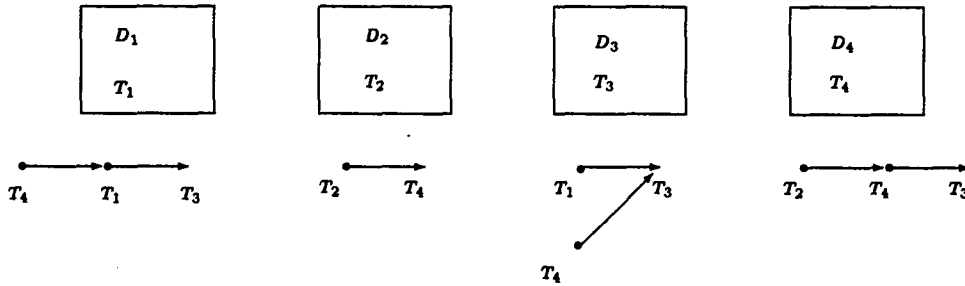


Figure 3.5: Condensed TWF Graphs

On the surface this algorithm appears reasonable, however, its informal specification hides both errors in logic and implementation complexities. These problems are discussed in [10]. Gligor and Shattuck outline the following counterexample that shows the incorrectness of the algorithm.

In this example transactions T_1 resides at site D_1 and has locked R_1 . T_2 has locked R_2 at site D_2 , and T_3 has locked R_3 at site D_3 . This initial configuration is shown in Figure 3.6. The following requests are then made; T_1 requests R_2 , T_2 requests R_3 , and T_3 requests R_1 . Figure 3.7 shows the portions of the graph that will be present at each site as a result of these requests. Each data manager will also send a blocking pair to the site where the requesting transaction resides.

Figure 3.8 shows the TWF graphs that will result when the messages containing the blocking pairs arrive. The protocol requires that new blocking pairs be generated if the

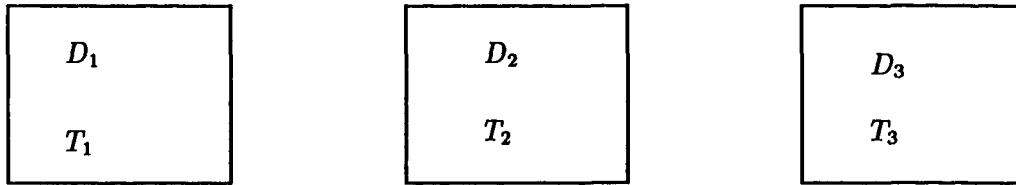


Figure 3.6: Counterexample: Initial Allocation

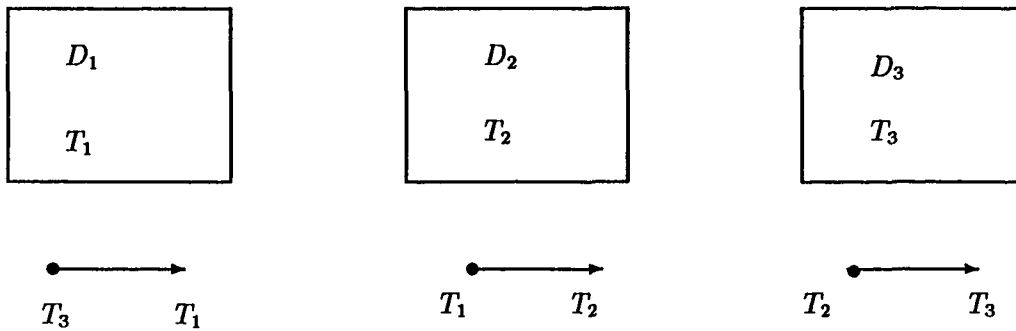


Figure 3.7: Counterexample: Second Phase

second element of any received pair is blocked in the new TWF graph. In this case when (T_1, T_2) arrives at D_1 , D_1 evaluates the new TWF graph and finds that T_2 is not blocked. Therefore, no further deadlock detection activity occurs at D_1 . When the blocking pairs arrive at D_2 and D_3 , arcs are added to the local graphs, but no new blocking pairs are generated. As a result deadlock will not be detected. The error in this case is caused by the timing of the arrival of the requests and the messages with the blocking pairs. If the pair (T_1, T_2) had arrived before the request from T_3 , the blocking pair (T_3, T_2) would have been generated and sent to T_2 and T_3 . The protocol would have worked correctly then, and deadlock would have been detected.

This example illustrates how the concurrency imposed by message passing, and the resultant non-determinism of event ordering can introduce subtle errors into distributed protocols. The more complicated the protocol, the more likely it is that timing errors will occur. The complexity of this protocol is a direct result of the fact that this protocol tries to maintain a picture of global state.

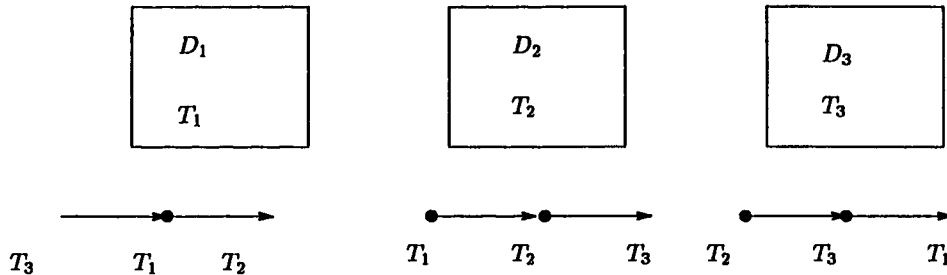


Figure 3.8: Counterexample: Deadlocked Set

Gligor and Shattuck suggest a modification to correct the particular error shown in the example, but they point out that the protocol is impractical even if it could be corrected. The real difficulty with the protocol occurs when resource releases and transaction aborts are considered. Menasce and Muntz neglect to describe how their protocol will maintain the condensed TWF graphs in the presence of releases and aborts. Because these activities cause the global state to be in a continual state of flux, extending their protocol to handle releases and aborts would make it more complicated. In addition, the overhead required to perform this correctly, to keep track of the new arcs and remove vestiges of the old arcs, is very high.

For example, consider what happens when a transaction releases a resource, and a new transaction is granted the resource. Every site which maintains an arc to the releasing transaction must be contacted so that the arc may be removed from the local TWF graphs. New arcs must be added for any transaction that now waits for the new transaction that has been granted the resource. Gligor and Shattuck suggest that a broadcast of all arc changes to all sites would be required to update the TWF graphs properly. This would be necessary because an individual data manager would not be aware of the sites that need updating. This continuous updating imposes high overhead on the system. It also makes the protocol complicated and prone to error.

All of these problems arise from the effort to maintain some approximation to a global TWF graph. The idea that a view of global state is necessary to solve the deadlock detection

problem is a holdover from the centralized multiprocessor environment. Other protocols which rely on the construction of TWF graphs have been proposed. For completeness we describe them briefly. However, we believe that a much simpler protocol, which is not predicated on some view of global state, is the appropriate solution to this problem.

Badal [9] approaches the distributed deadlock problem in a similar manner to Menasce and Muntz. Partial TWF graphs are maintained at resource sites. The primary difference in the two protocols is that, in Badal's algorithm, the transactions and their lock histories actually migrate to resource sites. This tends to consolidate state information and reduce the message passing necessary to update the graphs. Badal does not specify how he handles the problem of maintaining the graphs (he acts as if this is a solved problem) so it is difficult to evaluate his protocol with respect to some of the issues raised by Gligor and Shattuck. He acknowledges that false deadlocks can be detected because of obsolete information in the stored TWF graphs but claims that this is not a significant problem.

Elmagarmid, et.al.[7] also provide a variant of this type of algorithm. In their protocol, maintenance of the TWF graph is semi-centralized. This is accomplished by having requesting processors relinquish control to the processor holding the requested resource unless the holding transaction is also blocked. Any transaction making a request of a blocked transaction hands over control to the blocked transaction's controller. In this way any processes potentially involved in deadlock are arranged in a tree with one controlling node as the root. When a transaction relinquishes control it passes any information about wait-for relationships and any further resource requests to the controlling transaction. All the information necessary to detect deadlock is present in this root transaction. As all communication is done through the controlling transaction it always has an up-to-date view of the state of the controlled processes. The algorithm gets the remaining leverage it needs to eliminate the message timing problems found in other algorithms by being formulated in CSP.

All of the aforementioned algorithms are quite complex. We believe the complexity derives from the attachment to a global state view of the deadlock problem. This attachment

arises from the use of the TWF graph as a model for describing system behavior and designing deadlock detection protocols. A TWF graph implies a knowledge of global state. Efforts to construct these graphs are counter-productive in a distributed system because there is no global view in this environment. The token protocols we will discuss now show that construction of TWF graphs is not necessary to solve the deadlock detection problem.

Token, or probe, protocols as they tend to be called when dealing with deadlock detection, abandon the attempt to extend sequential system solutions to distributed systems. In these algorithms no explicit transaction-wait-for graphs are constructed or maintained. The token traverses the edges of the wait-for-graphs driven by blocked requests. The return of a token to its originator indicates that a cycle has been found. Chandy, Misra and Haas [5] propose a simple algorithm in which an idle process periodically generates a token to determine if it is part of a cycle. The token is transferred from one transaction to another when it is determined that a wait-for relationship holds. If a cycle exists the token will eventually return to its initiator, and deadlock will be detected.

This protocol does not completely abandon the concept of saving global state information. Each transaction maintains an array data structure which saves information about which processes are waiting for it. When a token from T_j arrives at transaction T_i , the j^{th} element of this array is filled in to indicate that T_j is dependent on T_i . This data structure, with its potential to preserve obsolete state information, makes deadlock resolution difficult. This is not a problem for the protocol as presented because it does not specify how deadlock resolution is to be performed.

This protocol also has some performance problems. First, a transaction may instigate several tokens per blocked request. Second, every transaction in the cycle may detect deadlock. This is not only imposes extra overhead messages, it makes it difficult to resolve deadlock.

Sinha and Natarajan [11] present a protocol that attempts to correct these deficiencies. In their protocol at most one transaction in a cycle will detect deadlock. This makes

resolution easier and reduces the number of tokens in transit. Their protocol also specifies that a transaction will generate only one token when it must wait for a resource. Finally, their protocol makes efficiency claims based on the fact that the token transmits state information that is saved and may be used in possible future deadlock detection activity. In comparison to [5] fewer messages will result in this protocol.

In an effort to make a simple token based protocol efficient, Sinha and Natarajan make it incorrect. In some circumstances, deadlocks will not be recognized because tokens are not always forwarded when necessary. Out-of-date information may be retained at some of the nodes in this algorithm resulting in detection of false deadlocks. These difficulties arise because the protocol is attempting to maintain pieces of the TWF graphs with information gleaned from the token. As we pointed out in our analysis of the protocol in [12], accurately maintaining this type of state information distributively is not easy. We will discuss this protocol in some detail because it is representative of the token based protocols. It also illustrates the unnecessary complexity that is introduced into a protocol by the attempt to get a grasp on global state.

Each transaction is assigned a priority based on the identification number assigned to the transaction. All transaction id numbers are unique, therefore, the priorities of the transactions are totally ordered. The protocol uses this order to minimize the number of token transmissions. The priority assignment is also used to identify a unique transaction that will detect deadlock, and instigate resolution.

In this protocol the data managers initiate the tokens used to detect deadlock. A data manager initiates a token in response to an *antagonistic conflict*. An *antagonistic conflict* exists when there is an outstanding request for a locked resource, and the transaction requesting the resource has a higher priority than the transaction that holds the resource. A data manager initiates a token if an antagonistic conflict is detected when a lock request arrives. Tokens are also generated when a resource is released and reallocated if there are requesting transactions in the queue that have higher priorities than the new holder of the

resource. The identity of the transaction that causes an antagonistic conflict is placed in the new token as the *initiator*. The data manager then transmits the token to the transaction that is currently holding the resource.

A transaction saves any token received in its data structure, *probe_Q*. If a transaction is waiting when it receives a token, it sends a copy of the token on to the data manager where it is waiting. When a transaction makes a lock request and waits for the lock to be granted, it sends a copy of its *probe_Q* to that data manager.

When a data manager receives a token, it compares the priority the transaction identified by the token to the priority of the transaction that is holding the resource. If the priority of the transaction in the token is less than the priority of the holder, the token is discarded. If the priority of the token transaction is higher, then the token is propagated to the holding transaction. If the priorities are identical, then deadlock has occurred, and the data manager initiates action to resolve the deadlock.

To resolve deadlock the data manager sends an abort message to the lowest priority transaction. This lowest priority transaction is designated the victim. The victim releases its locks and cancels any pending request. It then sends a clean message to the data manager where it is waiting, and aborts. This clean message is propagated around the cycle until it reaches the data manager that initiated the token. The clean message clears the tokens that reference the aborted victim from the *probe_Q* of each transaction as it passes. A transaction discards any clean message received if it is in active state or if it is the initiator.

Several advantages are claimed for this protocol. Each transaction initiates at most one token per blocked request. This differs from [5] which requires that tokens be periodically retransmitted. The use of transaction priorities and antagonistic conflicts guarantee that a unique token completes the cycle. This makes resolution easier. The information saved in the *probe_Q*'s can be used to detect later deadlocks with less work.

Saving information in the *probe_Q*'s enables the protocol to have these advantages. This information also causes difficulties. The information saved in the *probe_Q*'s is essentially state

information about the TWF graphs. As we pointed out in our analysis of the protocol in [12] it is hard to maintain this information accurately. These difficulties cause this protocol to be incorrect.

Several of these problems were identified by Choudhary, et. al., [8]. An error in the specifications for transmitting tokens allows deadlocks to exist which will never be detected by the protocol. The example shown in Figure 3.9 illustrates how this can occur. In this example, the token initiated because of T_1 is propagated to T_3 . R_3 is allocated to T_2 when it is released by T_3 . The protocol doesn't require that D_3 initiate a token when the reallocation is made to T_2 , because T_4 has a lower priority than T_2 . The protocol doesn't require that T_4 's *probe_Q* containing the token from T_1 be propagated on to T_2 . In the last step T_2 makes a request for R_6 which is held by T_1 . D_6 will not generate a token from T_2 because no antagonistic conflict exists. As a result deadlock will not be detected when it does exist.

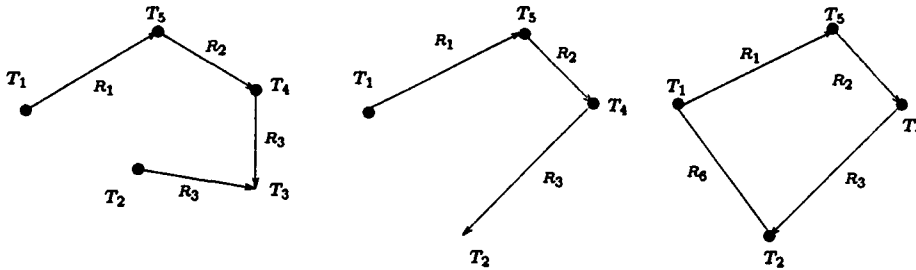


Figure 3.9: Undetected Deadlock

Tokens stored in the *probe_Q*'s of the transactions can lead to the detection of deadlocks that do not exist. The example shown in Figure 3.10, also from [8], exhibits how this protocol detects non-existent deadlocks.

Transactions T_2 and T_4 are deadlocked. Transaction T_1 waits transitively on transaction T_2 and T_4 . So the token from T_1 will be saved in the *probe_Q*'s of T_2 and T_4 . T_2 will detect the deadlock and cause T_4 to abort. After T_4 aborts you have the situation shown in the second TWF graph of Figure 3.10. T_4 will send a clean message to T_2 to remove any tokens that reference T_4 . The clean message from T_4 will not clean T_1 's token from T_2 's *probe_Q*.

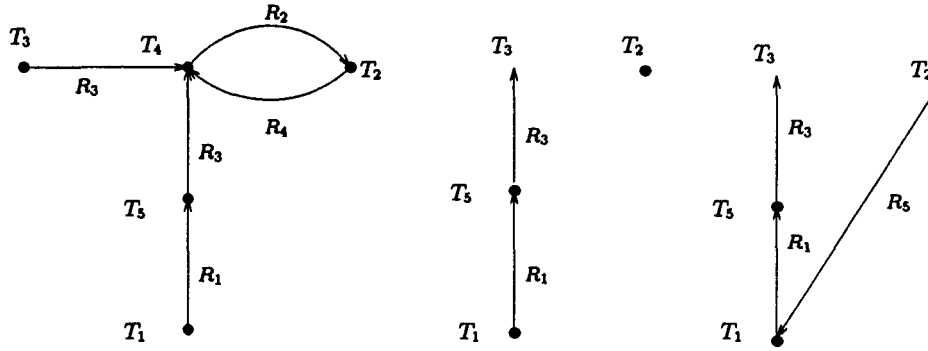


Figure 3.10: False Deadlock

The information that T_1 is waiting for T_2 is obsolete and can lead to detection of false deadlock. In the final TWF graph T_2 makes a request for R_1 , the resource held by T_1 . The data manager of R_1 will propagate the contents of T_2 's *probe_Q* to T_1 . Because this *probe_Q* contains a token from T_1 deadlock will be detected where none exists.

Choudhary, et.al. [8] recommend several changes to make Sinha and Natarajan's protocol correct. Their solution eliminates state information that has the potential to become obsolete. This requires that whenever an abort occurs all the *probe_Q*'s of transactions in the cycle must be discarded. Discarding the *probe_Q*'s necessitates that a token must be retransmitted for every transaction in the cycle and for every transaction waiting transitively on the cycle.

Their solution has several disadvantages. First, it introduces extra overhead because tokens and *probe_Q*'s must be retransmitted. Second, it negates most if not all of the performance benefits claimed in [11]. It can no longer be claimed that at most one token is generated per blocked request. Discarding the *probe_Q*'s means that detecting each deadlock must be done from scratch. Finally it adds complexity to an already complicated algorithm. This complexity and the informal specification of the protocol make it difficult to determine if the protocol is correct. Choudhary, et. al., do not attempt to show that the protocol is correct.

We believe that there is no need to maintain the kind of state information that characterizes the protocols in [8, 11, 12]. We have developed a causally based definition of deadlock. Using this definition we have designed a series of simple token based protocols that efficiently detect deadlock and provide resolution.

3.1.2 Deadlock and Causality

Deadlock is usually defined in global terms. For example, Chandy, Misra, and Haas[5] define deadlock as “a cycle of idle processes each dependent on the next process in the cycle.” Sinha and Natarajan [11] determine that deadlock occurs when “each member of the group waits (indefinitely) for a data item locked by some member transaction of the group”.

There is an implicit notion of global time in these definitions. As defined deadlock “exists” at a certain time, namely the time between when the last arc in the cycle falls into place and when the deadlock is broken. There is a similarity between this definition of deadlock and the standard definition of termination that we discussed previously. As we pointed out when discussing termination detection, when a global clock is absent, it obscures the issue to involve global time. It preserves the illusion that global state can be identified and used.

In termination detection this is primarily a theoretical problem. Viewing termination in causal terms clarifies the protocols and leads to better definition of the problem. As a matter of practice, however, existing termination protocols are relatively efficient and correct. In deadlock detection this global view has not only created theoretical difficulties, it also has caused the majority of existing protocols to be incorrect. Those that are correct are extremely complicated.

The reason for this is that deadlock detection and resolution differs from termination in one important way. Once processes are terminated, they remain terminated. The global state eventually quits changing, and in fact, an accurate view of the global state can eventually be constructed. In the case of deadlock, the object of detection is to resolve conflicts

so that deadlock is broken and transactions can proceed. Therefore, during execution of the protocol the global state is always changing. This makes it very difficult for deadlock detection protocols that attempt to base their actions on the global state.

A better approach is to define deadlock causally on a series of events and abandon the global view of deadlock. The probe protocols in [11, 5, 8] attempt to take this approach, but they can't quite resist the urge to tie protocol action to non-local state. However, there is a general pattern which occurs in the probe, or token, deadlock protocols which is useful for defining deadlock causally. This pattern occurs in the manner in which the transaction-wait-for graph is traversed. This is normally done by initiating a token at some blocked transaction and then passing it to the data manager of the object for which the transaction is waiting. The data manager, in turn, propagates the token to the transaction which holds the resource. The return of a token to its initiator indicates deadlock has occurred.

The use of a token generates a polling wave in a manner similar to the polling wave used for termination detection. The difference between the termination detection and deadlock detection polling waves is that in the first instance the token traverses a pre-determined path through all the processes in the system, while the deadlock detection token's path is determined dynamically based on the current relationships between transactions and data managers. Consequently, a deadlock detection polling wave may not encompass every process in the system.

A transaction generates a token to instigate a polling wave every time it must wait for a resource. The n^{th} token generated by T_i is identified as $tk(i, n)$. The set of processes visited by $tk(i, n)$ is designated by $S_i(n)$.

The following notation identifies the events of the deadlock detection polling wave:

Events of Detecting Computation

- $ct_j(i, n)$ the event which occurs when $tk(i, n)$ arrives at T_j .
- $wt_j(i, n)$ the event which occurs when $tk(i, n)$ leaves T_j .

- $cd_j(i, n)$ the event which occurs when $tk(i, n)$ arrives at D_j .
- $wd_j(i, n)$ the event which occurs when $tk(i, n)$ leaves D_j .

Using the following functions we can formally specify the token's behavior.

- $DW(T_i, e'_i) = D_j$ iff $\text{sendHold}_{D_j \rightarrow T_i}^k \rightarrow e'_i \wedge \text{recvGrant}_{D_j \rightarrow T_i}^k \not\rightarrow e'_i$
(D_j is the data manager of R_j . T_i is waiting for R_j at event e'_i).
- $HT(D_j, e'_j) = T_i$ iff $\text{recvGrant}_{D_j \rightarrow T_i}^k \rightarrow e'_j \wedge \text{recvRel}_{T_i \rightarrow D_j}^k \not\rightarrow e'_j$
(D_j is the data manager of R_j . T_i has been granted R_j and has not released it at event e'_j).

A polling wave is defined only for the events in the set $S_i(n)$. Events $wt_j(i, n)$ and $ct_j(i, n) \in PW(i, n)$ iff $T_j \in S_i(n)$. Similarly, $wd_j(i, n)$ and $cd_j(i, n) \in PW(i, n)$ iff $D_j \in S_i(n)$. A polling wave, $PW(i, n)$, is complete when $ct_i(i, n)$ occurs.

Token Specifications

- Token $tk(i, n)$ always initiated by T_i
 - If a token is propagated it moves from
 - T_j to $DW(T_j, ct_j(i, n))$, or
 - D_k to $HT(D_k, cd_k(i, n))$
-

In general, the order of the transactions and data managers visited is not fixed; i.e., the token will not necessarily visit T_1 then T_2 , etc. Therefore, $wt_j(i, n) \rightarrow cd_k(i, n)$ where $DW(T_j, wt_j(i, n)) = D_k$. Similarly $wd_j(i, n) \rightarrow ct_j(i, n)$ if $HT(D_j, wd_j(i, n)) = T_j$.

The following predicates formally define transaction and data manager states:

Predicates

- $Wait()$ is a function from events in a Transaction Process to $\{\text{True}, \text{False}\}$
- $Locked()$ is a function from events in a Data Manager Process to $\{\text{True}, \text{False}\}$

These predicates are defined as follows:

- $Wait(e'_i) = T$ iff $\exists j : \text{recvHold}_{D_j \rightarrow T_i}^k \rightarrow e'_i \wedge \text{recvGrant}_{D_j \rightarrow T_i}^k \not\rightarrow e'_i$.
- $Locked(e'_j) = T$ iff $\exists i : \text{sendGrant}_{D_j \rightarrow T_i}^k \rightarrow e'_j \wedge \text{recvRel}_{T_i \rightarrow D_j}^k \not\rightarrow e'_j$.

Using these predicates and our rules for token behavior, we can define deadlock in causal terms. A set of processes $S_i(n) = \{D_j \mid cd_j(i, n) \in PW(i, n)\} \cup \{T_k \mid ct_k(i, n) \in PW(i, n)\}$ is deadlocked if the polling wave $PW(i, n)$ completes and the following conditions hold:

Deadlock Conditions - DT

- DT(a)** $Wait(wt_j(i, n))$ for all $wt_j(i, n) \in PW(i, n) \wedge Locked(wd_j(i, n))$ for all $wd_j(i, n) \in PW(i, n)$.
- DT(b)** For all grant events, $\text{sendGrant}_{D_j \rightarrow T_k}^k$, such that $T_k, D_j \in S_i(n)$;
 $\text{sendGrant}_{D_j \rightarrow T_k}^k \rightarrow wd_j(i, n) \supset \text{recvGrant}_{D_j \rightarrow T_k}^k \rightarrow wt_k(i, n)$.
- DT(c)** For all release events, $\text{sendRel}_{T_k \rightarrow D_j}^k$, such that $T_k, D_j \in S_i(n)$;
 $\text{sendRel}_{T_k \rightarrow D_j}^k \rightarrow wt_k(i, n) \supset \text{recvRel}_{T_k \rightarrow D_j}^k \rightarrow wd_j(i, n)$.
-

The correctness conditions specified by DT are quite similar to those specified by T for termination. Because deadlock is a form of termination for a subset of processes in a system this is not surprising. Condition T(a) requires $Idle(wt_j(i, m))$ for each process polled. Condition DT(a) requires $Wait(wt_j(i, n))$ if the process polled is a transaction or $Locked(wd_j(i, n))$ if the process is a data manager. In both cases, the first conditions of DT and T require that a polled process is idle. Condition T(b) also corresponds to DT(b) and

DT(c) in that it requires that no communication between elements of the polled set are in transit during the wave.

These conditions guarantee that if a polling wave as specified completes, and the conditions are met, then deadlock exists at the “consistent cut” constructed by the wave. These conditions do not require that if a deadlock exists the protocol will detect it. For this aspect of correctness of a protocol we need to require that if deadlock exists, then eventually there will exist T_i such that $PW(i, m)$ completes. First, we will show that the following protocols satisfy these DT conditions. Then we will show that if deadlock does exist, then some polling wave will complete.

3.1.3 Synchronous Communication Protocols

Initially we will show how the availability of synchronized clocks provides for straightforward solution of this problem. We will then show how vector clocks can be readily substituted for real time clocks. As in termination detection we will show how the protocol and correctness arguments vary according to the system environment. Our first solution is designed for a system that provides synchronous communication. Initially we will also restrict our attention to situations where a transaction may have one outstanding request at a time.

We now describe a protocol which generates a polling wave for which DT holds. This protocol is modeled after Rana’s termination detection algorithm. Therefore, it presumes that global time is available, and that processes communicate synchronously. The protocol can also be viewed as a modification of the protocol proposed in [5]. In this modified protocol an idle process initiates a probe only once per blocked request. It also identifies a unique transaction to resolve deadlock. Initially we will only consider detection. For clarity we will postpone resolution to Section 3.1.5.

In this protocol a transaction T_i will generate a token when it receives a hold message. The m^{th} token generated by T_i will initiate $PW(i, m)$. The transaction places the timestamp

of the hold message receipt event in the token and sends it to the data manager of the resource the transaction is waiting for, $DW(T_i, wt_i(i, m))$. The data manager discards the token if the data resource is not locked, or if the timestamp of the latest grant message sent by the data manager is greater than the token timestamp. Otherwise the token is propagated by the data manager to the transaction that has locked the data manager's resource.

When a token arrives at a transaction it is discarded if the transaction is active. When a blocked transaction receives a token it compares the timestamp in the token to the timestamp of latest hold message receipt event. If the timestamp in the token is less than this timestamp, the token is discarded. On the other hand, if the timestamp in the token is greater than the timestamp in the process the token is propagated to the next data manager. There will, of necessity, be a transaction in a deadlocked set that receives the last hold message. This token will complete the circuit and deadlock will be detected.

Formally the token has two fields:

- $tk(i, m).ts$ = timestamp of latest *hd* event in T_i
- $tk(i, m).id$ = Transaction identifier

The following rules formally specify the protocol.

Deadlock Detection - Fully Synchronous System Protocol

SD.1 $wt_j(i, m), i \neq j$ occurs iff

$$\exists ct_j(i, m) \text{ such that } Wait(ct_j(i, m)) \wedge ct_j(i, m) \mapsto wt_j(i, m) \wedge (recvHold_{D_k \rightarrow T_j}^k \rightarrow ct_j(i, m) \supset T(recvHold_{D_k \rightarrow T_j}^k) < tk(i, m).ts).$$

SD.2 $wd_j(i, m)$ occurs iff

$$\exists cd_j(i, m) \text{ such that } Locked(cd_j(i, m)) \wedge cd_j(i, m) \mapsto wd_j(i, m) \wedge (sendGrant_{D_j \rightarrow T_k}^k \rightarrow cd_j(i, m) \supset T(sendGrant_{D_j \rightarrow T_k}^k) < tk(i, m).ts).$$

SD.3 $wt_j(j, m)$ occurs iff $\exists recvHold_{D_k \rightarrow T_j}^k$ such that $recvHold_{D_k \rightarrow T_j}^k \mapsto wt_j(j, m)$.

SD.4 The occurrence of $wt_j(j, m)$ implies

$$tk.(j, m).ts = T(\text{recvHold}_{D_k \rightarrow T_j}^k) \text{ where } \text{recvHold}_{D_k \rightarrow T_j}^k \mapsto wt_j(j, m) \wedge tk.(j, m).id = j.$$

SD.5 A polling wave, $PW(i, m)$ is complete when $ct_i(i, m)$ occurs.

Lemma 24 For any $wt_j(i, m)$ event as it is specified in the SD Protocol, $Wait(wt_j(i, m))$.

Proof: Rule SD.1 requires that $Wait(ct_j(i, m))$ be true. It also specifies that there does not exist e'_j such that $ct_j(i, m) \rightarrow e'_j \rightarrow wt_j(i, m)$. Therefore, $Wait(wt_j(i, m))$ must be true since there can be no receipt of a grant message between the receipt and transmission of the token at p_j . In the case of the generation of a token at $wt_j(j, m)$, clearly SD.3 guarantees that $Wait(wt_j(j, m))$ is true as well. ■

Lemma 25 For any $wd_j(i, m)$ event as it is specified in the SD Protocol, $Locked(wd_j(i, m))$.

Proof: Rule SD.2 requires that $Locked(cd_j(i, m))$ be true for any $cd_j(i, m) \in PW(i, m)$. Rule SD.2 also requires that there does not exist e'_j such that $cd_j(i, m) \rightarrow e'_j \rightarrow wd_j(i, m)$. Therefore, $Locked(wd_j(i, m))$ must be true. ■

In this protocol a token cannot return to its initiator unless the timestamp in the token is greater than the time of every `sendGrant` event that has occurred in the data managers that are traversed. Because communication is synchronous, the timestamp of any grant event, $\text{sendGrant}_{D_j \rightarrow T_i}^k$, must equal the timestamp of the corresponding $\text{recvGrant}_{D_j \rightarrow T_i}^k$ event. Therefore, the rules of the protocol also guarantee that the timestamp in the token is also greater than the timestamp of any `recvGrant` event. The fact that the token timestamp

exceeds every recvGrant event timestamp implies that each recvGrant event must happen before some polling event in the wave. Therefore, condition $\text{DT}(b)$ must hold. Lemma 26 proves formally that $\text{DT}(b)$ holds for the SD protocol.

Lemma 26 *If polling wave, $PW(i, m)$, as specified in the SD Protocol is complete then $\text{DT}(b)$ is true.*

Proof: If $\text{DT}(b)$ does not hold then there exists some $\text{sendGrant}_{D_j \rightarrow T_k}^k \rightarrow \text{wd}_j(i, m)$ such that $\text{recvGrant}_{D_j \rightarrow T_k}^k \not\rightarrow \text{wt}_k(i, m)$. $PW(i, m)$ is a complete wave. So, by SD.2, if $\text{sendGrant}_{D_j \rightarrow T_k}^k \rightarrow \text{wd}_j(i, m)$ then $T(\text{sendGrant}_{D_j \rightarrow T_k}^k) < \text{tk}(i, m).ts$. $T(\text{recvGrant}_{D_j \rightarrow T_k}^k) < \text{tk}(i, m).ts$ as well because communication is synchronous. The timestamp in the token must be less than any $\text{wt}_j(i, m) \in PW(i, m)$ events in the polling wave, therefore, $T(\text{recvGrant}_{D_j \rightarrow T_k}^k) < \text{tk}(i, m).ts < T(\text{wt}_k(i, m))$. This implies $\text{wt}_k(i, m) \not\rightarrow \text{recvGrant}_{D_j \rightarrow T_k}^k$. Since both events are in the same process, $\text{recvGrant}_{D_j \rightarrow T_k}^k \rightarrow \text{wt}_k(i, m)$, contradicting our original assumptions. ■

Lemma 27 *If polling wave, $PW(i, m)$, as specified in the SD Protocol is complete then $\text{DT}(c)$ is true.*

Proof: If $\text{DT}(c)$ doesn't hold then there exists some $\text{sendRel}_{T_k \rightarrow D_j}^k \rightarrow \text{wt}_k(i, m)$ such that $\text{recvRel}_{T_k \rightarrow D_j}^k \not\rightarrow \text{wd}_j(i, m)$. Two phase locking protocol requires that a transaction may not make any requests after releasing its resources. This contradicts Lemma 24 which shows that $\text{Wait}(\text{ct}_j(i, m))$. ■

Theorem 8 *The completion of a valid wave in the Deadlock Detection - Fully Synchronous System Protocol satisfies deadlock conditions $\text{DT}(a)$, $\text{DT}(b)$, and $\text{DT}(c)$.*

Proof: Directly follows from Lemmas 24, 25, 26, and 27. ■

In the case of termination detection we were able to derive a causal termination detection protocol by substituting a causal timestamp for a real timestamp in Rana's synchronous protocol. This technique is also useful in the case of deadlock detection. Instead of using the real time of the hold message event in the token we will use the vector timestamp of the *recvHold* event. Unlike the termination detection protocol, in this protocol the messages associated with the detecting computation affect the calculation of vector time just as any message in the underlying computation.

In the causal protocol each transaction, T_i , when it receives a hold message, generates a token to initiate $PW(i, m)$. The transaction places the vector timestamp of the *recvHold* event in the token. The token is sent to $DW(T_i, wt_i(i, m))$. The data manager discards the token if the data resource is not locked. The data manager will also discard the token if the vector timestamp of the latest grant message sent by the data manager is greater than the token timestamp. Otherwise, the token is propagated by the data manager to $HT(D_j, cd_j(i, m))$.

An active transaction discards the token. A blocked transaction compares the timestamp in the token to the vector timestamp of its latest *recvHold* event. If the timestamp in the token is greater than or concurrent to the timestamp of the *recvHold* event, the token is propagated to the next data manager. Otherwise the token is discarded.

The rules of the causal protocol are identical to the rules of the fully synchronous protocol except for the timestamp value in the token. The behavior is somewhat different because there will not be a unique token that completes a circuit. There may be several concurrent *recvHold* events, no one of which is "latest" in causal terms. There will, however, be at least one such transaction that will successfully detect deadlock.

Causal Deadlock Detection - Synchronous Communication

CSD.1 $wt_j(i, m), i \neq j$ occurs iff

$$\exists ct_j(i, m) \text{ such that } Wait(ct_j(i, m)) \wedge ct_j(i, m) \mapsto wt_j(i, m) \wedge \\ \text{recvHold}_{D_k \rightarrow T_j}^k \rightarrow ct_j(i, m) \supset V_j(\text{recvHold}_{D_k \rightarrow T_j}^k) \not\prec tk(i, m).ts.$$

CSD.2 $wd_j(i, m)$ occurs

$$\exists cd_j(i, m) \text{ such that } Locked(cd_j(i, m)) \wedge cd_j(i, m) \mapsto wd_j(i, m) \wedge \\ \text{sendGrant}_{D_j \rightarrow T_k}^k \rightarrow cd_j(i, m) \supset V_j(\text{sendGrant}_{D_j \rightarrow T_k}^k) \not\prec tk(i, m).ts.$$

CSD.3 $wt_j(j, m)$ occurs iff $\exists \text{recvHold}_{D_k \rightarrow T_j}^k$ such that $\text{recvHold}_{D_k \rightarrow T_j}^k \mapsto wt_j(j, m)$.

CSD.4. The occurrence of $wt_j(j, m)$ implies

$$tk.(j, m).ts = V_j(\text{recvHold}_{D_k \rightarrow T_j}^k) \text{ where } \text{recvHold}_{D_k \rightarrow T_j}^k \mapsto wt_j(j, m) \wedge \\ tk.(j, m).id = j.$$

CSD.5 A polling wave, $PW(i, m)$ is complete when $ct_i(i, m)$ occurs.

Lemma 28 For any $wt_j(i, m)$ and $wd_j(i, m)$ event as it is specified in the Causal Deadlock Detection - Synchronous Communication Protocol, $Wait(wt_j(i, m))$ and $Locked(wd_j(i, m))$.

Proof: See Lemma 24 and Lemma 25 ■

Lemma 29 If polling wave, $PW(i, m)$, as specified in the CSD Protocol is complete then $DT(b)$ is true.

Proof: If $DT(b)$ does not hold then there exists $\text{sendGrant}_{D_j \rightarrow T_k}^k$ such that $\text{sendGrant}_{D_j \rightarrow T_k}^k \rightarrow wd_j(i, m)$, and $\text{recvGrant}_{D_j \rightarrow T_k}^k \not\prec wt_k(i, m)$. $PW(i, m)$ is a complete wave. By Rule CSD.2,

$\text{sendGrant}_{D_j \rightarrow T_k}^k \rightarrow wd_j(i, m)$ implies $V_j(\text{sendGrant}_{D_j \rightarrow T_k}^k) < tk(i, m).ts$. $V_k(\text{recvGrant}_{D_j \rightarrow T_k}^k) < tk(i, m).ts$ as well because communication is synchronous. The timestamp in the token must be less than the timestamp of any wt event in the polling wave. Therefore, $V_k(\text{recvGrant}_{D_j \rightarrow T_k}^k) < tk(i, m).ts < V_k(wt_k(i, m))$. This implies $wt_k(i, m) \not\rightarrow \text{recvGrant}_{D_j \rightarrow T_k}^k$. Since both events are in the same process, $\text{recvGrant}_{D_j \rightarrow T_k}^k \rightarrow wt_k(i, m)$ contradicting our original assumptions. ■

Notice the striking similarity between the proofs of Lemmas 26 and 29. In fact, they are essentially identical.

The rules SD.1 and CSD.1 governing the transmission of a token from a transaction to a data manager are the same in the fully synchronous protocol and the causal protocol. This generic rule specifies that $wt_j(i, m)$ may not occur unless $\text{Wait}(ct_j(i, m))$. The requirements of two-phase locking in conjunction with this rule guarantees that $DT(c)$ cannot be violated regardless of whether a real timestamp, a vector timestamp, or no timestamp is present in the token.

Lemma 30 *If polling wave, $PW(i, m)$, as specified in the CSD Protocol is complete then $DT(c)$ is true.*

Proof: See Lemma 27. ■

Theorem 9 *The completion of a valid wave in the Causal Deadlock Detection - Synchronous Communication Protocol satisfies deadlock conditions $DT(a)$, $DT(b)$, and $DT(c)$.*

Proof: Follows directly from Lemmas 28, 29 and 30. ■

The deadlock conditions specified by DT guarantee that if the conditions are met then the set of transactions traversed by the token are in fact deadlocked at the polling wave events. The polling wave essentially identifies a set of system events for which deadlock can be said to hold. These conditions say nothing about whether a protocol that meets them will in fact detect any deadlock that occurs. The following theorems prove that if deadlock

occurs then some transaction in the deadlocked set will instigate a complete polling wave in the SD and CSD protocols.

Theorem 10 *If a set $S_j(x)$ is deadlocked then a complete wave $PW(j, x)$ as specified by SD will occur.*

Proof: If a set $S_j(x)$ is deadlocked then $\text{recvHold}_{D_k \rightarrow T_i}^k$ for some D_k has occurred for every transaction $T_i \in S_j(x)$. In a fully synchronous system the times of the recvHold events of the members of $S_j(x)$ are totally ordered. Consider T_j , where $T(\text{recvHold}_{D_m \rightarrow T_j}^k) > T(\text{recvHold}_{D_k \rightarrow T_i}^k)$ for all $T_i \in S_j(x)$, $i \neq j$. Rules SD.3 and SD.4 specify that T_j will generate a token, $tk(j, x)$ such that $tk(j, x).ts = T(\text{recvHold}_{D_m \rightarrow T_j}^k)$. $T(\text{recvHold}_{D_k \rightarrow T_i}^k) < T(wt_j(j, x))$, and $T(wt_j(j, x)) < T(ct_i(j, x))$, for all $T_i \in S_j(x)$. Therefore, $\text{recvHold}_{D_k \rightarrow T_i}^k \rightarrow ct_i(j, x)$. Because $tk(j, x).ts > T(\text{recvHold}_{D_k \rightarrow T_i}^k)$, and $\text{Wait}(ct_i(j, x))$, for all $T_i \in S_j(x)$, Rule SD.1 will be satisfied for every $T_i \in S_j(x)$, $i \neq j$.

$T(\text{recvHold}_{D_k \rightarrow T_i}^k) < tk(j, x).ts$ implies $T(\text{sendHold}_{D_k \rightarrow T_i}^k) < tk(j, x).ts$, for all transactions in the deadlocked set. The latest hold message event in D_k must be preceded by the latest grant message in D_k . Therefore, for all l , $T(\text{sendGrant}_{D_k \rightarrow T_l}^k) < tk(j, x).ts$, and $T(\text{sendGrant}_{D_k \rightarrow T_l}^k) < tk(j, x).ts < T(wt_j(j, x))$. This implies that $\text{sendGrant}_{D_k \rightarrow T_l}^k \rightarrow cd_k(j, x)$, and $\text{Locked}(cd_k(j, x))$. Rule SD.2 will be satisfied for each $D_k \in S_j(x)$.

Because $S_j(x)$ is deadlocked a cycle will exist. The token will eventually return to T_j , $ct_j(j, x)$ will occur, and the $PW(j, x)$ will be complete (Rule SD.5). ■

A similar argument can be made for the causal protocol. The proof differs somewhat because the vector timestamps of the recvHold events are not totally ordered as the physical timestamps are in a fully synchronous system

Theorem 11 *If a set $S_j(x)$ is deadlocked then a complete wave $PW(j, x)$ as specified by CSD will occur.*

Proof: If a set $S_j(x)$ is deadlocked then $\text{recvHold}_{D_k \rightarrow T_i}^k$ for some D_k has occurred for every

transaction $T_i \in S_j(x)$. In an asynchronous system, the vector timestamps of the `recvHold` events of the members of $S_j(x)$ are partially ordered. There will be a set of one or more transactions such that the timestamps of the members of this set will be greater than, or concurrent to, the timestamps of every member of $S_j(x)$. Consider one member of this set, T_j , where $V_j(\text{recvHold}_{D_m \rightarrow T_j}^k) \not\prec V_i(\text{recvHold}_{D_k \rightarrow T_i}^k)$ for all $T_i \in S_j(x)$, $i \neq j$. Rules CSD.3 and CSD.4 specify that T_j will generate a token, $tk(j, x)$ such that $tk(j, x).ts = V_j(\text{recvHold}_{D_m \rightarrow T_j}^k)$. $\text{recvHold}_{D_m \rightarrow T_j}^k \rightarrow wt_j(j, x)$, so $V_j(\text{recvHold}_{D_m \rightarrow T_j}^k) < V_j(wt_j(j, x))$. Therefore, for all $T_i \in S_j(x)$, $V_j(wt_j(j, x)) < V_i(ct_i(j, x))$. Therefore, $\text{recvHold}_{D_k \rightarrow T_i}^k \rightarrow ct_i(j, x)$. Because $tk(j, x).ts \not\prec V_i(\text{recvHold}_{D_k \rightarrow T_i}^k)$, and $Wait(ct_i(j, x))$ for all $T_i \in S_j(x)$, Rule CSD.1 will be satisfied for every $T_i \in S_j(x)$, $i \neq j$.

If $tk(j, x).ts \not\prec V_i(\text{recvHold}_{D_k \rightarrow T_i}^k)$ for all transactions in the deadlocked set, then $tk(j, x).ts \not\prec T(\text{sendHold}_{D_k \rightarrow T_i}^k)$. The latest hold message event in D_k must be preceded by the latest grant message in D_k . Therefore, for all i , $tk(j, x).ts \not\prec V_k(\text{sendGrant}_{D_k \rightarrow T_i}^k)$. This implies that $\text{sendGrant}_{D_k \rightarrow T_i}^k \rightarrow cd_k(j, x)$. Otherwise $tk(j, x).ts < V_k(\text{sendGrant}_{D_k \rightarrow T_i}^k)$. Therefore $Locked(cd_k(j, m))$. Rule CSD.2 will be satisfied for each $D_k \in S_j(x)$.

Because $S_j(x)$ is deadlocked a cycle will exist. The token will eventually return to T_j , $ct_j(j, x)$ will occur, and the $PW(j, x)$ will be complete (Rule CSD.5). ■

3.1.4 Asynchronous FIFO Communication Protocols

Synchronous communication imposes a performance burden on a distributed system. Having it available can simplify the description of the problem and its solution. In termination detection as we relaxed the communication restrictions the protocols became more complex. In deadlock detection it turns out that there is no benefit from synchronous communication. The protocols described in Section 3.1.3 do not change when defined for a system that allows asynchronous communications. The requirement that well ordered message passing be preserved is still necessary to design straightforward protocols.

The following lemmas illustrate the only changes in the proofs that are required to show

that SD and CSD perform correctly in a system with asynchronous FIFO communications. The lemmas apply without change to both SD and CSD.

Lemma 31 *If polling wave, $PW(i, m)$, as specified in the SD or CSD Protocol is complete then $DT(b)$ is true.*

Proof: If $DT(b)$ does not hold then there exists $\text{sendGrant}_{D_j \rightarrow T_k}^k$ such that $\text{sendGrant}_{D_j \rightarrow T_k}^k \rightarrow wd_j(i, m)$ and $\text{recvGrant}_{D_j \rightarrow T_k}^k \not\rightarrow wt_k(i, m)$. By Rule CSD.2, $wd_j(i, m)$ occurs if and only if $\text{Locked}(cd_j(i, m))$. This implies $\text{recvRel}_{T_k \rightarrow D_j}^k \not\rightarrow cd_j(i, m)$ and $HT(D_j) = T_k$. The rules that govern the token's travel require $cd_j(i, m) \rightarrow wd_j(i, m) \rightarrow ct_k(i, m)$. Since communication is FIFO, and $\text{sendGrant}_{D_j \rightarrow T_k}^k \rightarrow wd_j(i, m)$, it must be the case that $\text{sendGrant}_{D_j \rightarrow T_k}^k \rightarrow \text{recvGrant}_{D_j \rightarrow T_k}^k \rightarrow ct_j(i, m)$, contradicting our original assumption. ■

Lemma 32 *If polling wave, $PW(i, m)$, as specified in the SD or CSD Protocol is complete then $DT(c)$ is true.*

Proof: See Lemma 27. ■

Theorem 12 *The completion of a valid wave in the SD or CSD Protocols satisfies deadlock conditions $DT(a)$, $DT(b)$, and $DT(c)$.*

Proof: Follows directly from Lemmas 28, 31 and 32. ■

3.1.5 Deadlock Resolution

The preceding arguments have treated deadlock as a static condition. with many features in common with termination. Deadlock differs from termination in one important aspect. The point of identifying a set of deadlocked processes is to break the deadlock so that the computation may proceed. The standard method used to resolve the deadlock is to force

one of the transactions in the deadlocked set to abort. When a process aborts it releases any resource it holds, and any pending requests are cancelled. One way to resolve deadlock would be to abort all the processes involved in the deadlock. This wouldn't be the most efficient solution. To minimize the performance impact of this procedure most protocols attempt to abort a single process in the deadlock cycle.

One obvious way to accomplish this is to structure the protocol so that only one member of the deadlock cycle detects deadlock. Then that member may either abort itself, or direct some other member of the cycle to abort. Sinha and Natarajan use transaction identification numbers to designate, *a priori*, the unique transaction in a cycle that will detect deadlock. Because there is no assurance that the deadlock cycle will be complete when the token from the highest priority transaction begins its traversal, it is necessary to save a record of this token in the *probe_Q*'s of the transactions traversed by the token. Otherwise record of the token would be lost when an active transaction is encountered. We have noted the difficulty caused by saving this state information.

In a system that has a global clock it is easy to identify a unique process to detect deadlock. The token timestamps of all the transactions in a deadlocked set are totally ordered. By specifying that the latest token prevails, the fully synchronous protocol guarantees that deadlock is detected by a unique process.

In our causal protocol it is possible that more than one process will detect deadlock because of the concurrency of the timestamps of the hold replies. It is necessary to add information other than the timestamp to the token to insure that a unique process is chosen to be aborted. There are two possibilities. First, as the token travels through the deadlocked set the process id of each transaction is collected. Each transaction that detects deadlock will collect an identical set of these transaction id's. When a polling wave completes, the detecting transaction will send an abort message to some member of the deadlocked set based on some predetermined criteria, such as lowest id number. This will potentially result in several processes sending an abort message. However, a unique process will abort.

A second possibility that utilizes the vector timestamps of the transactions and does not require the transmission of any abort messages, requires that the token collect process ids for any transaction whose timestamp is concurrent to the timestamp in the token. In this manner, each transaction that detects deadlock will be aware of the other transactions which could detect deadlock. Based on its process id a transaction that detects deadlock may then decide whether to abort or not unilaterally. So, if transaction T_k detects deadlock, and the returning token contains transaction id's $k+1$ and $k+x$, then T_k knows that it must abort. On the other hand, if any transaction id in the token is less than k , then transaction T_k knows that some other member of the deadlocked set will detect the deadlock and abort.

Our causal definition of deadlock simplifies the resolution process. The other protocols we discussed ran into difficulty when attempting to resolve deadlock because non-local state information was retained at the processes. Aborting a transaction required cleaning up any state information that referred to the aborted transaction. This cleanup activity imposes overhead on the system and complexity on the protocol. In these causal protocols a self-designated process aborts. There is no need for messages to go to an abort victim, and there is no need to clean up obsolete state information. Because interaction between a transaction and a data manager during an abort is very similar to that which occurs during a regular release, major changes in the protocol are not needed to perform resolution. Consequently, the correctness arguments for these protocols need little modification.

The first protocol we will present that performs deadlock resolution is a modification of the Fully Synchronous System protocol. This protocol uses synchronized real time clocks and allows asynchronous, FIFO communication. The transaction with the latest timestamp detects deadlock and aborts. The second protocol we will present is a causal protocol which substitutes vector clocks for real time clocks in the synchronous deadlock detection and resolution protocol. Because there may not be a unique latest timestamp, the token must be modified to carry extra information.

Before we formally define these modified protocols, it is necessary to define appropriate

abort events.

- $\text{sendAbort}_{T_i \rightarrow D_j}$ - transmission of an abort request from T_i to D_j
- $\text{recvAbort}_{T_i \rightarrow D_j}$ - receipt of abort request from T_i at D_j
- as_i - start event of abort phase in T_i

When a transaction, T_i , is in an abort phase it will send release messages to the data managers responsible for the resources T_i holds. A sendAbort message will also be sent to the data manager responsible for the resource for which T_i is waiting. Once these messages have been sent, T_i will terminate.

We now define the following predicates:

- $\text{Abort}(e_i)$ iff $as_i \rightarrow e_i$
- $\text{Queue}_j(T_i, e'_j)$ iff $\text{sendHold}_{D_j \rightarrow T_i}^k \rightarrow e'_j \wedge \text{sendGrant}_{D_j \rightarrow T_i}^k \not\rightarrow e'_j \wedge \text{recvAbort}_{D_i \rightarrow T_j} \not\rightarrow e'_j$

Synchronous Deadlock Detection and Resolution - Asynchronous FIFO Communication

SDR.1 $wt_j(i, m), i \neq j$ occurs iff

$$\exists ct_j(i, m) \text{ such that } \text{Wait}(ct_j(i, m)) \wedge ct_j(i, m) \mapsto wt_j(i, m) \wedge \text{recvHold}_{D_k \rightarrow T_j}^k \rightarrow ct_j(i, m) \supset T(\text{recvHold}_{D_k \rightarrow T_j}^k) < tk(i, m).ts \wedge \neg \text{Abort}(ct_j(i, m)).$$

SDR.2 $wd_j(i, m)$ occurs

$$\exists cd_j(i, m) \text{ such that } \text{Locked}(cd_j(i, m)) \wedge cd_j(i, m) \mapsto wd_j(i, m) \wedge \text{sendGrant}_{D_j \rightarrow T_k}^k \rightarrow cd_j(i, m) \supset T(\text{sendGrant}_{D_j \rightarrow T_k}^k) < tk(i, m).ts.$$

SDR.3 $wt_j(j, m)$ occurs iff $\exists \text{recvHold}_{D_j \rightarrow T_k}^k$ such that $\text{recvHold}_{D_j \rightarrow T_k}^k \mapsto wt_j(j, m)$.

SDR.4 The occurrence of $wt_j(j, m)$ implies

$$tk.(j, m).ts = T(\text{recvHold}_{D_k \rightarrow T_j}^k) \text{ where } \text{recvHold}_{D_k \rightarrow T_j}^k \mapsto wt_j(j, m) \wedge tk.(j, m).id = j.$$

SDR.5 A polling wave, $PW(i, m)$ is complete when $ct_i(i, m)$ occurs.

SDR.7 as_i occurs iff $ct_i(i, m)$ occurs.

SDR.8 The occurrence of as_i implies

$$\forall j \text{ such that } HT(D_j, wd_j(i, m)) = T_i, as_i \rightarrow \text{sendRel}_{T_i \rightarrow D_j}^k \wedge \\ \text{if } \text{sendHold}_{D_j \rightarrow T_i}^k \rightarrow ct_i(i, m) \text{ then } as_i \rightarrow \text{sendAbort}_{T_i \rightarrow D_j}.$$

We now show that this modified protocol meets the deadlock conditions specified in DT.

Lemma 33 *For any $wt_j(i, m)$ and $wd_j(i, m)$ event as it is specified in the Synchronous Deadlock Detection and Resolution - Asynchronous FIFO Communication Protocol; $Wait(wt_j(i, m))$, and $Locked(wd_j(i, m))$.*

Proof: Neither Rule SD.1 or Rule SD.2 are modified in SDR so that $\neg Locked(wd_j(i, m))$, or $\neg Wait(wt_j(i, m))$, for any polling event. Therefore DT(a) is still satisfied. ■

Lemma 34 *If polling wave, $PW(i, m)$, as specified in the SDR Protocol is complete then $DT(b)$ is true.*

Proof: The argument used in Lemma 31 to show DT(b) is dependent on the requirements of Rule SD.2 and the rules governing the behavior of the token. These rules are unaffected by the modifications of SDR, therefore DT(b) must hold. ■

Lemma 35 *If polling wave, $PW(i, m)$, as specified in the SDR Protocol is complete then $DT(c)$ is true.*

Proof: During normal execution of a transaction, Rule SD.1 and the requirements of two-phase locking guarantees that $\text{sendRel}_{T_k \rightarrow D_j}^k \not\rightarrow wt_k(i, m)$ for any transaction T_k . Therefore, DT(c) holds when a transaction completes normally. Rule SDR.1 is modified to accommodate aborting transactions. This rule specifies that $wt_j(i, m)$ can occur if and only if $\neg Abort(ct_j(i, m))$. This means that it is not possible for $as_i \rightarrow wt_j(i, m)$. Any release

instigated by the abort occurs after as_i , by Rule SDR.8. Therefore, it is impossible for $as_i \rightarrow \text{sendRel}_{T_j \rightarrow D_k}^k \rightarrow wt_j(i, m)$ for any k . Hence, $DT(c)$ must hold. ■

Theorem 13 *The completion of a valid wave in the Synchronous Deadlock Detection - Asynchronous FIFO Communication Protocol satisfies deadlock conditions $DT(a)$, $DT(b)$, and $DT(c)$.*

Proof: Directly follows from Lemmas 33, 34, and 35. ■

The token used in the following causal protocol has three fields:

- $tk(i, m).ts$ = timestamp of latest *hd* event in T_i
- $tk(i, m).id$ = Transaction identifier
- $tk(i, m).set$ = set of transaction identifiers

Causal Deadlock Detection and Resolution - Asynchronous FIFO Communication

CSDR.1 $wt_j(i, m), i \neq j$ occurs iff

$$\exists ct_j(i, m) \text{ such that } Wait(ct_j(i, m)) \wedge ct_j(i, m) \mapsto wt_j(i, m) \wedge \\ \text{recvHold}_{D_k \rightarrow T_j}^k \rightarrow ct_j(i, m) \supset V_j(\text{recvHold}_{D_k \rightarrow T_j}^k \not\prec tk(i, m).ts. \wedge \\ \neg Abort(ct_j(i, m))).$$

CSDR.2 $wd_j(i, m)$ occurs

$$\exists cd_j(i, m) \text{ such that } Locked(cd_j(i, m)) \wedge cd_j(i, m) \mapsto wd_j(i, m) \wedge \\ \text{sendGrant}_{D_j \rightarrow T_k}^k \rightarrow cd_j(i, m) \supset V_j(\text{sendGrant}_{D_j \rightarrow T_k}^k \not\prec tk(i, m).ts).$$

CSDR.3 $wt_j(j, m)$ occurs iff $\exists \text{recvHold}_{D_k \rightarrow T_j}^k$ such that $\text{recvHold}_{D_k \rightarrow T_j}^k \mapsto wt_j(j, m)$.

CSDR.4. The occurrence of $wt_j(j, m)$ implies

$$tk.(j, m).ts = V_j(\text{recvHold}_{D_k \rightarrow T_j}^k) \text{ where } \text{recvHold}_{D_k \rightarrow T_j}^k \mapsto wt_j(j, m) \wedge \\ tk.(j, m).id = j \wedge \\ tk.(j, m).set = \emptyset.$$

CSDR.5 A polling wave, $PW(i, m)$ is complete when $ct_i(i, m)$ occurs.

CSDR.6 The occurrence of $wt_j(i, m)$ where $V_j(\text{recvHold}_{D_k \rightarrow T_j}^k) \parallel tk(i, m).ts$ implies

$$tk(i, m).set = tk(i, m).set \cup j.$$

CSDR.7 as_i occurs iff

$$ct_i(i, m) \text{ occurs } \wedge \\ i < j \text{ for all } j \in tk(i, m).set.$$

CSDR.8 The occurrence of as_i implies

$$\forall j \text{ such that } HT(D_j, wd_j(i, m)) = T_i, as_i \rightarrow \text{sendRel}_{T_i \rightarrow D_j}^k \wedge \\ \text{if } \text{sendHold}_{D_j \rightarrow T_i}^k \rightarrow ct_i(i, m) \text{ then } as_i \rightarrow \text{sendAbort}_{T_i \rightarrow D_j}.$$

The same arguments used to show that SDR satisfies DT apply to CSDR.

Theorem 14 *The completion of a valid wave in the Causal Deadlock Detection and Resolution - Asynchronous FIFO Communication Protocol satisfies deadlock conditions $DT(a)$, $DT(b)$, and $DT(c)$.*

Proof: Directly follows from Lemmas 33, 34, and 35. ■

It is now necessary to show that execution of these modified protocols result in detection and resolution of deadlock. Theorem 15 shows that the desired result holds for the synchronous protocol SDR.

Theorem 15 *If a set $S_j(x)$ is deadlocked then a complete wave $PW(j, x)$ as specified by SDR will occur, and deadlock of $S_j(x)$ will be resolved.*

Proof: If a set $S_j(x)$ is deadlocked then $\text{recvHold}_{D_k \rightarrow T_i}^k$ for some D_k has occurred for every transaction $T_i \in S$. Consider T_j , where $T(\text{recvHold}_{D_m \rightarrow T_j}^k) > T(\text{recvHold}_{D_k \rightarrow T_i}^k)$ for all $T_i \in S_j(x)$, $i \neq j$. T_j will generate a token, $tk(j, x)$ such that $tk(j, x).ts = T(\text{recvHold}_{D_m \rightarrow T_j}^k)$ (Rules SDR.3 and SDR.4). $T(RH_{D_m} T_j) < T(ct_i(j, x))$ for all $T_i \in S_j(x)$.

Therefore, $\text{recvHold}_{D_k \rightarrow T_i}^k \rightarrow \text{ct}_i(j, x)$, and $\text{Wait}(\text{ct}_i(j, x))$. For Rule SDR.1 to be satisfied for all $T_i \in S_j(x)$, $i \neq j$, $\neg \text{Abort}(\text{ct}_i(j, x))$ must also hold. $\text{Abort}(\text{ct}_i(j, x))$ implies $\text{ct}_i(i, y) \rightarrow \text{ct}_i(j, x)$, by Rule SDR.7. This implies that the latest recvHold event in T_i , $\text{recvHold}_{D_k \rightarrow T_i}^k$, must have a later timestamp than $\text{recvHold}_{D_m \rightarrow T_j}^k$. Contradicting our hypothesis about $T(\text{recvHold}_{D_m \rightarrow T_j}^k)$. Because $tk(j, x).ts > T(\text{recvHold}_{D_k \rightarrow T_i}^k)$, $\text{Wait}(\text{ct}_i(j, x))$, and $\neg \text{Abort}(\text{ct}_i(j, x))$ for all $T_i \in S_j(x)$, Rule SDR.1 will be satisfied for every $T_i \in S_j(x)$, $i \neq j$.

The remainder of the argument is the same as that given in Theorem 10. We repeat it here for completeness.

If $T(\text{recvHold}_{D_k \rightarrow T_i}^k) < tk(j, x).ts$ holds for all transactions in the deadlocked set, then $T(\text{sendHold}_{D_k \rightarrow T_i}^k) < tk(j, x).ts$. The latest hold message event in D_k must be preceded by the latest grant message in D_k . Therefore, for all l , $T(\text{sendGrant}_{D_k \rightarrow T_l}^k) < tk(j, x).ts$, and $T(\text{sendGrant}_{D_k \rightarrow T_l}^k) < tk(j, x).ts < T(wt_j(j, x))$. This implies that $\text{sendGrant}_{D_k \rightarrow T_l}^k \rightarrow \text{cd}_k(j, x)$, and $\text{Locked}(\text{cd}_k(j, x))$. Therefore, Rule SDR.2 will be satisfied for each $D_k \in S_j(x)$.

Because $S_j(x)$ is deadlocked a cycle will exist, and the token will eventually return to T_j . By Rule SDR.5, $\text{ct}_j(j, x)$ will occur, $PW(j, x)$ will be complete, and deadlock will be detected. Completion of $PW(j, x)$ implies the occurrence of as_j , the release of resources held by T_j , and the removal of T_j from the queue of the data manager of the resource for which T_j is waiting (Rules SDR.6 and SDR.7). Therefore, the deadlock detected by $PW(j, x)$ will be broken. ■

The following argument that the CSDR protocol will detect deadlock parallels Theorem 11 very closely.

Theorem 16 *If a set $S_j(x)$ is deadlocked then a complete wave $PW(j, x)$ as specified by CSDR will occur, and deadlock of $S_j(x)$ will be resolved.*

Proof: If a set $S_j(x)$ is deadlocked then $\text{recvHold}_{D_k \rightarrow T_i}^k$ for some D_k has occurred for

every transaction $T_i \in S_j(x)$. In an asynchronous system the vector timestamps of the `recvHold` events of the members of $S_j(x)$ are partially ordered. There will be a set, $S'_j(x)$ of one or more transactions such that the timestamps of the members of this set will be greater than, or concurrent to, the timestamps of every member of $S_j(x)$. Consider the member of this set, T_j , such that $j < i$ for all $T_i \in S'_j(x)$, $i \neq j$. T_j will generate a token, $tk(j, x)$ such that $tk(j, x).ts = V_j(\text{recvHold}_{D_m \rightarrow T_j}^k)$ (Rules CSDR.3 and CSDR.4). $\text{recvHold}_{D_m \rightarrow T_j}^k \rightarrow wt_j(j, x)$, so $V_j(\text{recvHold}_{D_m \rightarrow T_j}^k) < V_j(wt_j(j, x))$. Therefore, for all $T_i \in S_j(x)$, $V_j(\text{recvHold}_{D_m \rightarrow T_j}^k) < V_j(wt_j(j, x)) < V_i(ct_i(j, x))$, and $\text{recvHold}_{D_k \rightarrow T_i}^k \rightarrow ct_i(j, x)$, and $\text{Wait}(ct_i(j, x))$. For Rule CSDR.1 to be satisfied for all $T_i \in S_j(x)$, $i \neq j$, $\neg \text{Abort}(ct_i(j, x))$ must also hold. $\text{Abort}(ct_i(j, x))$ implies $ct_i(i, y) \rightarrow ct_i(j, x)$, by Rule CSDR.7. This implies that the latest `recvHold` event in T_i , $\text{recvHold}_{D_k \rightarrow T_i}^k$, must have a timestamp concurrent to $\text{recvHold}_{D_m \rightarrow T_j}^k$, and $i < j$. This contradicts our original assumptions. Therefore, $tk(j, x).ts \not\prec V_i(\text{recvHold}_{D_k \rightarrow T_i}^k)$, and $\text{Wait}(ct_i(j, x)) \wedge \neg \text{Abort}(ct_i(j, x))$ for all $T_i \in S$. Rule CSDR.1 will be satisfied for every $T_i \in S_j(x)$, $i \neq j$.

If $tk(j, x).ts \not\prec V_i(\text{recvHold}_{D_k \rightarrow T_i}^k)$ holds for all transactions in the deadlocked set, then $tk(j, x).ts \not\prec V_i(\text{sendHold}_{D_k \rightarrow T_i}^k)$. The latest hold message event in D_k must be preceded by the latest grant message in D_k . Therefore, for all l , $tk(j, x).ts \not\prec V_k(\text{sendGrant}_{D_k \rightarrow T_l}^k)$. This implies that $\text{sendGrant}_{D_k \rightarrow T_l}^k \rightarrow cd_k(j, x)$. Otherwise $tk(j, x).ts < V_k(\text{sendGrant}_{D_k \rightarrow T_l}^k)$. Therefore $\text{Locked}(cd_k(j, m))$. Rule CSDR.2 will be satisfied for each $D_k \in S_j(x)$.

Because $S_j(x)$ is deadlocked a cycle will exist, and the token will eventually return to T_j . By Rule CSDR.5, $ct_j(j, x)$ will occur, $PW(j, x)$ will be complete, and deadlock will be detected. Completion of $PW(j, x)$ implies the occurrence of as_j , the release of resources held by T_j , and the removal of T_j from the queue of the data manager of the resource for which T_j is waiting (Rules CSDR.6 and CSDR.7). Therefore, the deadlock detected by $PW(j, x)$ will be broken. ■

The causal protocols we presented in this chapter illustrate the advantage of using local state and causal reasoning to analyze and solve dynamic distributed computational

problems. In Chapter 4 we will consider the problem of optimistic recovery. In contrast to the problems of distributed termination and deadlock detection, consideration of the causal order of events has always been part of the solution to the problem of optimistic recovery. The interesting aspect of this problem is the effect that process failure and rollback has on the causal relation.

Chapter 4

Optimistic Recovery

4.1 Introduction

An important consideration in the design of distributed systems is how to make them resilient to failure. The use of checkpoints on stable storage and rollback-recovery protocols are well established techniques for dealing with process failures within a distributed system [40, 41, 42, 43, 44]. When failure occurs a rollback protocol uses checkpoints and message logs to return the system to a *consistent* global state. By *consistent* global state we mean that if the receipt of a message has been recorded in the state of some process then the event of sending that message must also be recorded.

The strategies used in existing rollback-recovery protocols fall into three broad categories. Pure checkpointing schemes use process synchronization or some variation of global snapshots to construct a series of consistent checkpoints to be used as a basis for recovery [40, 45, 41]. Pessimistic message logging protocols require that each message is logged to stable storage when it is received [42, 43]. This technique makes recovery easier, but it extracts a performance penalty because each communication must be synchronized with the log operation. Optimistic message logging protocols avoid the need for synchronization by taking checkpoints asynchronously and logging messages asynchronously with commu-

nication. This method complicates the recovery process because messages can be lost when a process fails. The potential loss of messages, due to failure, makes it necessary to identify dependency relationships between process states induced by communication, so that a consistent system state can be constructed after failure [44, 46, 47, 48].

We will only consider optimistic recovery here. The synchronization required by pure check-pointing and pessimistic message logging protocols makes the order of events irrelevant to the design of such protocols. On the other hand, optimistic recovery is only possible because causal dependence imposes a partial order on events. Because identifying the causal partial order is essential to performing optimistic recovery, vector clocks are useful in designing these protocols. In this chapter we will present three causally based protocols which use vector clocks to perform optimistic recovery.

4.2 Previous Research

Restoration of a distributed computation after failure requires the use of information saved to stable storage by checkpoints of process state and logging of messages. This information is used to recover as much execution history as possible in the failed process. Strom and Yemini [44] developed the first rollback-recovery protocol that did not require synchronization during check-pointing or synchronization of message logging with communication. In their protocol processes make periodic checkpoints of their state onto stable storage. The check-pointing activity of one process occurs independently of the actions of other processes. The receipt of incoming messages is also logged to volatile storage, and periodically the message logs are moved to stable storage. Optimistic recovery is characterized by this asynchronous use of stable storage. Saving state and message logs to stable storage asynchronously reduces the performance impact of the rollback-recovery protocol on the failure-free execution of the system. However, more work is required to restore the computation after failure. Presumably, the time between failures is large, and we seldom incur

the extra restoration expense.

In a system operating under an optimistic recovery protocol, a failed process, upon recovery from failure, has its state restored to the state saved in the latest checkpoint in stable storage. The message logs saved to stable storage are used to reconstruct the process execution that occurs after the restored checkpoint. Because message logging to stable storage is not synchronized with message receipt, some incoming messages may not be in the stable logs after recovery. If this is the case, it is not possible to reconstruct the entire execution history of the failed process from the logs in stable storage. As far as the recovery process is concerned, any event that cannot be recovered using the stable logs never occurred. As a result, the state of the recovering process may be inconsistent with the states of the non-failed processes. This inconsistency occurs if any non-failed process has received a message from the failed process, and the corresponding transmission is not part of the recovered execution history.

One of the main activities of the rollback-recovery protocol is to return the system to a consistent state. This is done by identifying *orphan messages* and *orphan states*. A message is an orphan if it has been received by a non-failed process, but the record of its transmission has been lost by the sender due to failure. Any process state that causally depends on an orphan message is considered an orphan state. The existence of orphan states causes the global state of the distributed computation to be inconsistent. To return the system to a consistent state, the effect of the orphan messages must be eliminated from the execution of the system by “rolling back” the state of each non-failed process to before the occurrence of an orphan state.

Strom and Yemini use causal dependence to identify the states that must be eliminated through rollback. The system model they use to define causal dependence employs state intervals to describe process execution. A process history is divided by the receipt of messages in the process. In this model the receipt of the n^{th} message in a process p_i begins state interval s_i^n . The notation s_i^n signifies the n^{th} state interval in p_i . During a state

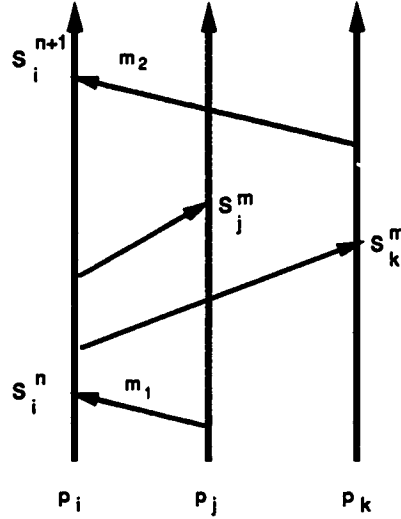


Figure 4.1: State Intervals

interval, p_i may or may not send messages to other processes. State s_i^n persists until the next message is received by p_i , causing the state to change to s_i^{n+1} . Figure 4.1 illustrates this system model. The receipt of message m_1 begins s_i^n , the n^{th} state interval in p_i . State interval s_i^{n+1} begins with the arrival of m_2 .

Their rollback algorithm determines which states must be rolled back in case of failure by identifying dependency relationships between process state intervals. State interval s_i^n is said to *depend* on s_j^m , $s_j^m \prec s_i^n$, if p_i receives a message from p_j which was sent during s_j^m , and the message begins s_i^n . Referring again to Figure 4.1, $s_j^{m-1} \prec s_i^n$, because of the message m_1 . Depends is a transitive relation so in this example $s_j^{m-1} \prec s_i^n$, and $s_i^n \prec s_k^m$, therefore, $s_j^{m-1} \prec s_k^m$.

When failure occurs in a process, the state interval begun by an unlogged message cannot be recovered from the information in stable storage. These state intervals are identified as orphan state intervals, and the messages that were sent during orphaned state intervals are orphan messages. If s_i^n is an orphaned state interval in failed process p_i , then any state interval s'_k in an active process for which $s_i^n \prec s'_k$ is also orphaned. Their protocol uses the

depends relation to identify and rollback orphaned state intervals.

The protocol specifies that a vector of state interval indices called a dependency vector is maintained by each process. The dependency vector is attached to every outgoing message. The dependency vector attached to an incoming message is used by the process receiving the message to update its own dependency vector. The receipt of a message by process p_i causes p_i to increment the i^{th} index of its dependency vector by one. Each of the remaining indices are set to the maximum of the current index of the local vector and the incoming index of the dependency vector attached to the message.

The dependency vectors maintained in each process and transmitted with each message are used to identify the causal relationships between process states as defined by " \prec ". By comparing the dependency vector of one state interval to the dependency vector of another state interval, the protocol can determine whether the depends relation holds between the two states. In this way dependency vectors can be used to identify orphan states.

When a process recovers from failure, it sends a recovery message, containing the index of the latest state interval that it is able to recover from stable storage, to every other process. Non-failed processes can determine if their state causally depends on one of the states lost by the failed process by comparing the state interval index in the recovery message to the corresponding index in its own dependency vector. If the state interval index associated with the failed process in the dependency vector of the non-failed process is greater than the last recovered state interval index communicated in the recovery message, then the current state of the non-failed process has been orphaned and must be eliminated through rollback.

When a process rolls back it retrieves the latest check-pointed state that is not an orphan. The latest non-orphan checkpoint is found by comparing the dependency vectors saved in the checkpoints to the state interval index in the recovery message. Once the checkpoint has been restored, messages are replayed from the message log until an orphan message is reached. At this point, the process acts as if it has failed and sends a recovery message containing its last recoverable state index to every other process. The system is

returned to a consistent state when every process has rolled back to a non-orphan state.

In a failure-free system each state interval index of a process is unique. The isomorphism between the causal partial order, “ \prec ”, and the ordering of dependency vectors depends on the uniqueness of the state interval indices. In this protocol, process failure and rollback result in the reuse of state interval indices. Consequently, in a system recovering from failure the isomorphism may no longer hold, and dependency vectors alone are not sufficient to identify the causal relationship between process states.

For example, consider a process, p_i , which fails during state interval s_k . The state of p_i is recovered, up to and including s_{k-m} , through the use of checkpoints and message logs. When p_i resumes execution, it will pass through state intervals $s_{k-m+1}, s_{k-m+2}, \dots$. A message sent by p_i before failure from state interval s_{k-m+1} will have the same i^{th} index in its attached dependency vector as a message sent by p_i during the s_{k-m+1} state interval after recovery. The dependency vectors attached to these messages cannot be used to identify the states on which these messages causally depend. The first message causally depends on a state that no longer exists, and is, therefore, an orphan. The second message causally depends on an active state, and therefore, is a valid message. Because the i^{th} state index in the attached dependency vectors are identical, it is impossible to distinguish between them.

It is possible for messages originating in orphan states to be in transit during the recovery process. If such a message arrives at a process that has already rolled back, it must be discarded, or it will cause the system to become inconsistent. On the other hand a correct protocol should not discard any non-orphan messages. Therefore, it is necessary for the protocol to be able to distinguish between “slow” orphan messages and valid messages. Because of duplicated state interval indices, the dependency vectors attached to messages do not convey enough information to make this distinction.

Strom and Yemini deal with this problem by assigning an *incarnation number* to every process. Every time a process restarts after failure or rolls back it must increment its

incarnation number. The dependency vector actually becomes a vector of ordered pairs

$$\langle (\iota_0, \mu_0), (\iota_1, \mu_1), \dots, (\iota_{N-1}, \mu_{N-1}) \rangle,$$

where ι_i is the incarnation number of p_i during state interval μ_i . Each process maintains an incarnation start table which tracks the first state interval for each incarnation of each process. Using this table and the dependency vector of an incoming message, orphan messages can be distinguished from non-orphan messages.

Once rollback is complete, the recovery process must insure that non-orphan messages that are lost due to failure and rollback are regenerated. Strom and Yemini's algorithm accomplishes this with a third set of indices that tracks the expected sequence number of incoming messages on a channel. If a process receives a message number that is higher than expected, some messages have been lost, and the process requests their retransmission from the sender. In some cases this method will not result in the regeneration of lost messages. Figure 4.2 illustrates a case where this occurs. Part (a) of the figure shows the failure of p_0 after sending message m_3 . Part (b) shows the system after it has returned to a consistent state. If p_1 never sends another message to p_0 , p_0 will not be made aware that message m_1 was lost and will not request its retransmission.

Strom and Yemini's protocol illustrates the importance of causal dependence to optimistic recovery. It also shows how dependency vectors are useful for identifying the relevant causal relationships. However, process failure and rollback destroys the isomorphism between dependency vectors and the causal partial order, thus limiting the usefulness of the dependency vectors. The addition of per-process incarnation numbers to the dependency vectors is necessary to overcome this difficulty.

The use of per-process incarnation numbers provides an additional benefit in that it enables this protocol to accommodate multiple failures, however, it also impairs performance. Because each process must be made aware of every new incarnation number, each process

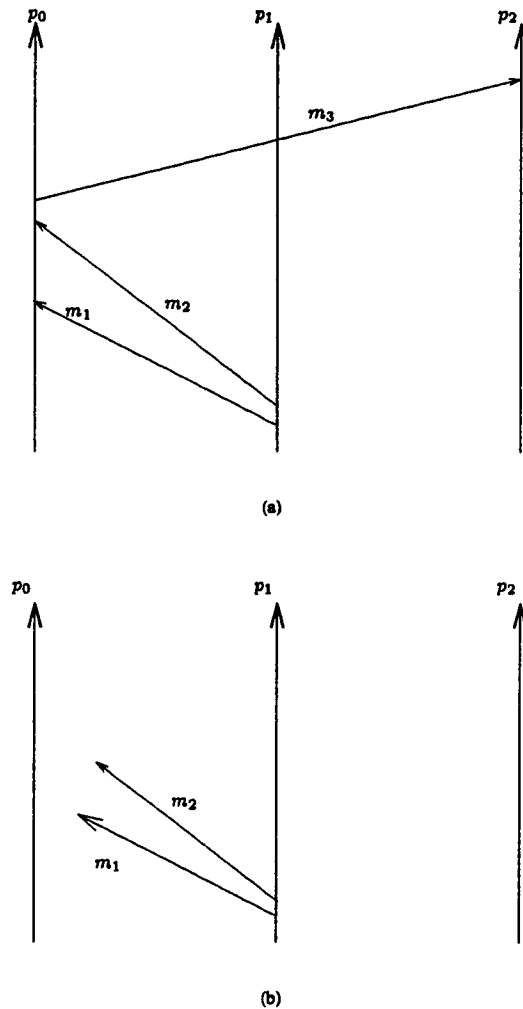


Figure 4.2: Example - Failure to Recognize Lost Messages

must communicate with every other process during rollback. Another result of augmenting the dependency vector with individual incarnation numbers is that only the (state interval index; incarnation number) pair can be used to determine the state to which a process should roll back. This can lead to multiple rollbacks per failure. To see this consider the sample execution shown in Figure 4.3.

Process p_0 sends two messages, m_1 and m_2 before failure. Because no checkpoint was

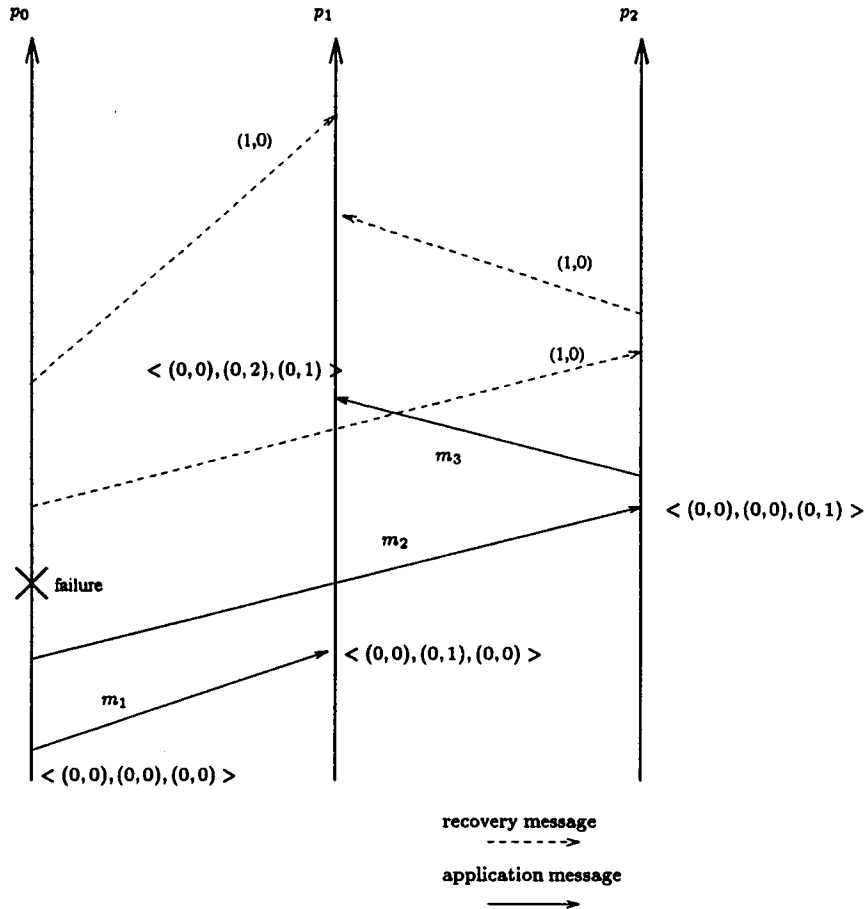


Figure 4.3: Example - Multiple Rollback

taken in p_0 , the record of transmission of m_1 and m_2 cannot be recovered from stable storage. When it recovers from the failure, p_0 sends two recovery messages containing the latest state interval recovered and its current incarnation number. In this case, p_0 recovers state interval s_0^0 and sets its current incarnation number to 1. The recovery message to p_2 causes p_2 to roll back to before the receipt of m_2 . p_2 will send a recovery message to p_1 containing the pair, $(1,0)$, indicating that it has rolled back to state interval s_1^0 . Upon the receipt of this recovery message p_1 will roll back before the receipt of m_3 . The recovery

message from p_2 will not cause p_1 to roll back before the receipt of m_1 , even though the state of p_1 after the receipt of m_1 is orphaned by the same failure that caused p_2 to roll back.

When p_1 receives the recovery message from p_0 , it will roll back again. The multiple rollback of p_1 occurs because p_2 only communicates its own (state interval index; incarnation number) pair in its recovery message. Unfortunately, to maintain the per-process incarnation number correctly, this is a necessary restriction on process behavior.

Sistla and Welch [47] developed an optimistic recovery protocol that also uses the state interval model and dependency vectors composed of state interval indices to identify orphan states. In their approach, a process, upon restart after failure, communicates its last recovered state index to every other process. When a non-failed process receives a recovery message from a failed process, it sends the state interval of the latest state that does not causally depend on the lost state of the failed process to every other process. Each process gathers the dependency information from all other process into a local vector. When the vector is completed, the process replays messages from the stable log until they run out, or a message with an attached dependency vector greater than the vector constructed during rollback appears. As the messages are replayed the process repeats its execution, including the sending of any messages that were previously sent. Duplicate messages are discarded upon arrival. A duplicate message is identified by comparing the dependency vector attached to the message to the value of the receiving process' current dependency vector.

The communication of every process with every other process accomplishes two purposes. First, a consistent state is recovered based on the dependency vectors constructed at each site. Second, all orphan messages are flushed out of the system before recovery is complete by the transmission of recovery messages along every channel. This eliminates the need for incarnation numbers to be used for distinguishing orphan messages from non-orphan messages. Lost messages are recovered by having each rolled back process retransmit all

messages. Dependency vectors are used to identify duplicate messages and discard them.

Both of the protocols discussed here have their own strengths and weaknesses. Strom and Yemini's protocol will accommodate multiple failures during the recovery process. This does not come without a cost. Their technique has the potential to cause multiple rollbacks per failure which would cause a high message overhead. In the best case, $O(N^2)$ messages are required per failure. Sistla and Welch's protocol can guarantee that no process rolls back more than once per failure. However, their protocol can not handle concurrent failures, and it also requires $O(N^2)$ messages per failure. Both protocols require the ongoing screening of all incoming messages. Strom and Yemini's protocol must identify and discard incoming orphan messages. Sistla and Welch must identify and discard duplicate messages. Sistla and Welch's protocol is structured in such a way as to not need incarnation numbers and incarnation start tables, but this is done at the cost of freezing each process until it has heard from every other process during recovery.

It is difficult to determine whether either of these protocols work correctly. In [44] verbal description is used to specify the actions of the protocol. This informality makes it hard to identify exactly what the protocol is supposed to do and whether it performs these functions correctly. Such informal specifications lead to errors in the protocol such as we illustrated in Figure 4.2. Sistla and Welch use pseudo-code and Input-Output automata to specify their protocol. While this is an improvement over the informal descriptions used in [44], it is still not clear that the protocol performs as specified.

The correctness arguments in both [44, 47] deal with the global behavior of the system. As we have shown when discussing deadlock, it is difficult to make reasoned arguments about the global behavior of a distributed system based on the actions of individual processes. As a consequence global arguments often appear to be correct, but are in fact wrong. The error we detected in Strom and Yemeni's protocol is an example of how a global argument is inadequate. We believe it is more defensible to argue about the local behavior of a process and use that to insure the correctness of a distributed protocol. In our presentation of

termination detection and deadlock detection, we used causality and vector time to define necessary “local” correctness criteria and prove that the protocol met those criteria.

This would seem to be an ideal solution in this problem as well. However, as we shall show in Section 4.3, causality as it is normally defined does not apply in a system subject to failure and rollback. The isomorphism between the causal partial order and vector time is not assured either. Without a clear understanding of the impact of failure on causality, it is not possible to use it to formally solve this problem. Our object here is to develop a meaningful definition of causality in a failure prone system and use it as a unifying construct. We will use it to formally define the criteria for judging a protocol to be correct. We will use it to develop axiomatic protocol specifications, and finally, we will use it to argue the correctness of our protocol in a straightforward manner.

4.3 Rollback and Causality

There is an obvious parallel between the \prec relation and Lamport’s happened before relation. The primary difference is that \rightarrow is defined for atomic events such as a message receipt or send, while the depends relation applies to a set of events. The effect of this difference is that time increases only when a message receipt occurs. Therefore, $s_i^n \prec s_i^m$ implies $s_i^n \rightarrow s_i^m$ if the logical clock values used to determine \rightarrow are only assigned to message receipts.

Conceptually the relations are very similar. However, Lamport’s happened before relation is widely recognized and immediately associated with the concept of causality in a distributed system. Lamport’s definition of causality has the extra advantage of having a formalized relation to vector time, namely the isomorphism between the clock value order and the causal partial order. For this reason we chose to model process behavior with atomic events and use \rightarrow instead of the \prec relation used in existing research.

The following example illustrates the steps needed to restore a system to a consistent state after failure when the system model is based on atomic events. In Figure 4.4(a)

process p_0 fails following the transmission of message m_3 . Figure 4.4(b) shows the state of the system following the recovery of p_0 . The state of p_0 was restored to its value at checkpoint ck_0^1 . Message m_1 has been replayed from the stable log. Process p_0 is unaware that it had received message m_2 and had sent message m_3 . The resulting system state is inconsistent because p_1 shows the receipt of m_3 , but p_0 does not have a record of the transmission of this message. The state of p_1 must be rolled back to eliminate the receipt of m_3 . In addition, any event that causally depends on m_3 , such as the transmission and receipt of m_4 , must also be eliminated. Figure 4.4(c) shows the consistent system state that results from rolling back p_1 and p_2 .

The existence of inconsistent states in the system as a result of process failure implies the disruption of the causal partial order imposed by Lamport's \rightarrow relation. In Figure 4.4 the partial order has been corrupted by the loss of m_1 and m_2 . Restoring the system to a consistent state requires that the causal partial order also be restored, so that the causal relationship between events conforms to Lamport's definition. One way to look at the rollback-recovery problem is to require that the system be returned to a consistent state. We prefer to view a correct protocol as one that restores the partial order. Therefore, we define the rollback problem in terms of the partial order and develop our protocols accordingly.

We will present several protocols that use causal dependence and vector timestamps to solve this problem. The first protocol we present requires $O(N)$ messages per recovery and does not require that any process wait for any other process during recovery. Its disadvantage is that it uses incarnation numbers and will not tolerate concurrent failure. Our second protocol is also of order N , but it is strictly causally based in that it doesn't require the use of incarnation numbers. It also will not tolerate concurrent failure. Our final protocol will tolerate multiple failure, but requires more synchronization during recovery.

Before we present our protocols we define some system parameters and notation.

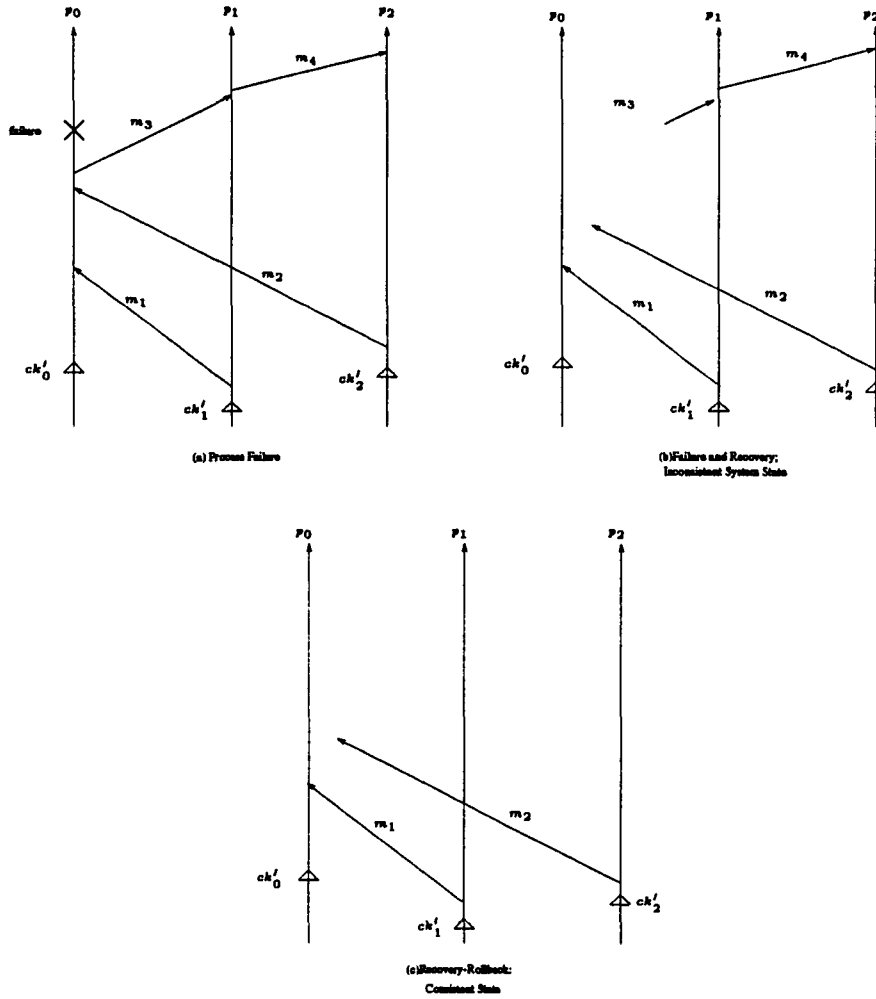


Figure 4.4: Failure and Rollback

4.3.1 Generic Model - Terminology

- p_i : Process i , which is part of the N process distributed application $\Pi = \{p_0, p_i, \dots, p_{N-1}\}$.
- e_j^i : The i^{th} event of p_j . Exactly what constitutes an event is specific to an application, but the transmission and receipt of application messages are always considered events. We use e_j' and e_j'' to refer to generic events of p_j .
- s : A send event of the underlying computation.

- $\eta(s)$: The receive event which matches transmission s .
- $\sigma(s)$: The process where send event s occurs.
- $\rho(s)$: The process where the receive event matched with send event s occurs.
- f_j^i : The i^{th} failure event on p_j .

We assume the system has the following characteristics:

Crash Failures - A failed process sends no messages, receives no messages, and performs no local state transitions.

Reliable FIFO channels - All messages between two processes are received in the order sent. All transmitted messages are received after an arbitrary but finite length of time. In addition, no messages are corrupted or duplicated.

Stable storage - Information saved in stable storage must be recoverable after failure.

Volatile storage - Information saved only in volatile storage is lost by process failure.

We also require that each processor knows its successor in a logical circuit of Π ; that knowledge must be in stable storage, since it is critical that it be recoverable after a failure.

Without loss of generality, we assume that $p_{(i+1) \bmod N}$ is the successor of p_i for $0 \leq i < N$.

4.3.2 Historical Causality

Causal dependence in a distributed system has come to be synonymous with Lamport's happened before relation. This is defined as the smallest relation such that:

1. If event e'_i and e''_i are in the same process, and e'_i occurs before e''_i , then $e'_i \rightarrow e''_i$;
2. If an event e'_i is the sending of a message in process p_i and e'_j is the receipt of this message in process p_j , then $e'_i \rightarrow e'_j$;

3. If $e'_i \rightarrow e'_j$ and $e'_j \rightarrow e'_k$, then $e'_i \rightarrow e'_k$.

Rule 1 of this definition specifies the causal relation between two events in the same process. This rule states that the temporal order of two such events determines their causal order. One of the assumptions underlying this rule is that events never “disappear” from the execution history of a process. In a failure-free system this is a valid assumption. Process failure and rollback can cause the loss of events from a process execution history, thus violating this basic assumption. As a consequence, Lamport’s causal relation no longer accurately identifies potential causality between events in a system that uses rollback to recover from failure. The following discussion of process behavior during failure, recovery, and rollback illustrates the impact of event loss on the commonly defined causal relationships between events. The events, functions, and predicates defined below are used to formalize this discussion.

- ck_j^i : The i^{th} state checkpoint on p_j . The checkpoint resides on stable storage.
- rs_j^i : The i^{th} restart event on p_j .
- rb_j^i : The i^{th} rollback event on p_j .
- $Latest.ck(f_j^i)$: The most recent checkpoint before f_j^i . $Latest.ck(f_j^i) = ck_j^i$ if $ck_j^i \rightarrow f_j^i \wedge \nexists ck_j''$ such that $ck_j^i \rightarrow ck_j'' \rightarrow f_j^i$.
- $LastEvent(f_j^i) = e'_j$ iff $e'_j \rightarrow rs_j^i$ and there does not exist e''_j such that $e'_j \rightarrow e''_j \rightarrow rs_j^i$.
- $CK(ck_j^i, e'_j)$ iff the state of p_j at e'_j is written to stable storage during checkpoint ck_j^i .
- $Logged(e'_j)$ iff e'_j is logged to stable storage

When process p_i , is restored after failure f'_i , its state is set to $Latest.ck(f'_i)$. The application is re-executed in p_i using the checkpoint state and the messages which had been logged to stable storage. A restart event associated with f'_i occurs when this process is completed. Any event in p_i which cannot be recovered from stable storage is *lost*.

Figure 4.5 shows the execution history of a failed process. $\text{Latest.ck}(f_i^m) = ck_i^1$. To recover, the state of p_i is set to ck_i^1 . In this example, suppose events e_i^2, e_i^3 , and e_i^4 are recoverable from stable storage. Any event which occurred after the last recovered event in p_i is lost (events e_i^5 and e_i^6). $\text{LastEvent}(f_i^m) = e_i^4$, because it is the last event to be replayed from the stable logs.

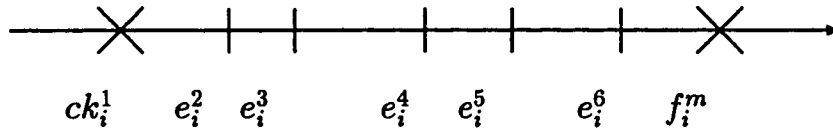


Figure 4.5: Failed Execution History

Figure 4.6 Part(a) shows the actual sequence of events as they occur over time. According to this temporal order and the rules defining the happened before relation, $e_i^6 \rightarrow rs_i^m$. The loss of e_i^6 from the execution history of p_i (as shown in Figure 4.6) invalidates this relationship. Event e_i^6 may precede rs_i^m in time, but it makes no sense to say that e_i^6 causally precedes rs_i^m .

Part (b) of Figure 4.6 is a more accurate depiction of the causal order. It illustrates how any event not saved to stable storage is removed from the causal partial order by the failure f_i^m . These events that are lost from the current set of events have no causal relationship to any events in the current set of events.

A similar disruption of the causal partial order in a single process may occur as a result of events being eliminated through rollback. When a process is rolled back, the current process state is discarded. A new process state is constructed from an earlier checkpoint and some subset of the messages saved in volatile and stable storage. When rollback is completed a rollback event, rb_i' occurs. As a result of the rollback procedure, some events may be discarded from stable and volatile storage. As in the failed process these discarded events are *lost*.

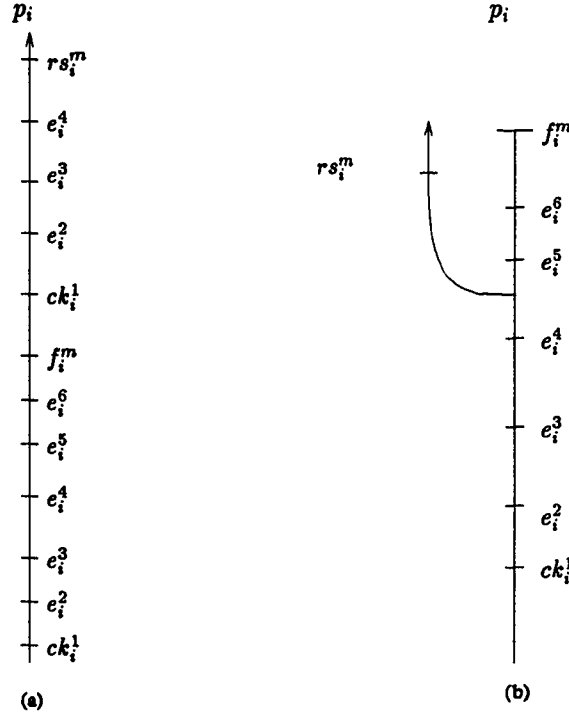


Figure 4.6: Temporal and Causal History

Figure 4.7 shows the effects of rollback on the execution history of a process. In this example, events e_j^{10} and e_j^{11} must be eliminated through rollback. The state of p_j is instantiated to the state saved at checkpoint ck_j^1 . Events e_j^8 and e_j^9 are replayed. Rollback event rb_j' occurs when the process is completed.

The effect of e_j^{10} and e_j^{11} have been removed from the execution history of p_j by process rollback. Therefore, even though e_j^{10} and e_j^{11} occur before rb_j' in time, they can have no causal effect on rb_j' .

These examples show that the loss of events due to failure and rollback invalidate the use of the temporal order of events as the sole basis for the causal order of events in one process. For this reason, the first rule in Lamport's definition of causality must be altered to account for the destruction of the causal links that occur when process events are lost.

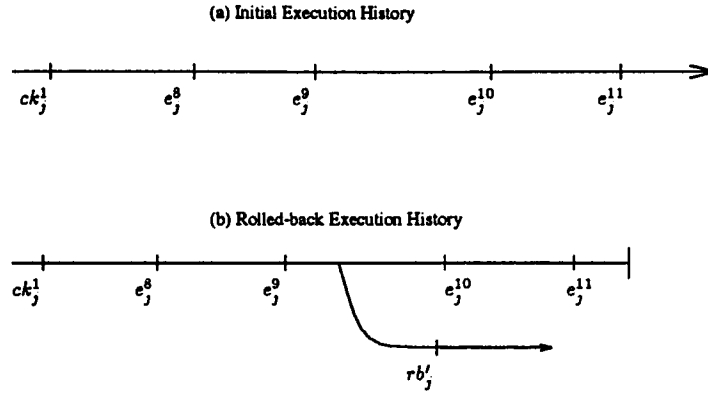


Figure 4.7: Effect of Rollback on Causal Order

In the preceding discussion of the impact of failure and rollback on the execution of a process we have characterized a process's execution history as a set of events. The current state of a process is determined by its initial state and the set of currently observable events that have occurred in that process. For a process, p_i , this set is denoted $e_current_i$. If a process has recovered from a failure or has been rolled back there may be events which have occurred in p_i , but are no longer in the current execution history. This set of events is e_lost_i . Given that e'_i is the latest event in $e_current_i$, e_lost_i can be defined as follows: $e''_i \in e_lost_i$ if and only if $e''_i \rightarrow e'_i$, and $e''_i \notin e_current_i$. The set, $e_all_i = e_lost_i \cup e_current_i$, is the set of all events that have occurred in p_i .

The set of events that define the system execution can be constructed from the event sets of the individual processes. So that,

- $E_current = \bigcup_{i=0}^{N-1} e_current_i$
- $E_lost = \bigcup_{i=0}^{N-1} e_lost_i$
- $E_all = E_lost \cup E_current$

Using the set $E_current$, we define a new "temporary" causality relation, called the *historical causality relation* and denoted by \Rightarrow , as the smallest binary relation for which

1. If $e'_i \rightarrow e''_i$, $e'_i \in E_current$, and $e''_i \in E_current$, then $e'_i \Rightarrow e''_i$;
2. If an event e'_i is the sending of a message in process p_i , and e'_j is the receipt of this message in process p_j , and $e'_j \in E_current$, then $e'_i \Rightarrow e'_j$;
3. If $e'_i \Rightarrow e'_j$, and $e'_j \Rightarrow e'_k$, then $e'_i \Rightarrow e'_k$.

Note that the second rule defining “ \Rightarrow ” is also a modification of the second rule in the definition of \rightarrow . This is because this aspect of causal precedence is only affected by lost events in the receiving process. When a process receives a message, the causal impact of the event of sending that message persists, even if the record of sending the message is lost by the sender. For example, the system execution shown in Part (a) of Figure 4.8 shows the events that occurred before failure f_i^m . Part (b) of Figure 4.8 shows the system state after of recovery of p_i .

Event e_i^4 is lost from the execution history of p_i . However, the causal effect of e_i^4 is still manifest in p_j , so $e_i^4 \Rightarrow e_j^6$, and $e_i^4 \Rightarrow e_j^7$ still hold. Due to the transitive nature of \Rightarrow , $e_i^4 \Rightarrow e_k^2$ as well. These causal relationships between lost events and events at non-failed processes cause the system to be inconsistent after failure. The inconsistency arises because $e_i^4 \Rightarrow e_j^7$, but $e_i^4 \notin E_current$. When one of these inconsistent events is identified and eliminated through rollback, it is removed from the current event set, and the causal relationship between the send and receive no longer holds. The second rule defining the causal order must be modified to recognize this fact. The system is returned to consistency when all the events that are members of the \Rightarrow relation are also elements of $E_current$.

The number of events in $E_current$ will increase and decrease as execution proceeds, and failures and rollbacks occur. For this reason this causal relation is temporary. Referring to Figure 4.8 again, at the time of failure, $e_i^4 \Rightarrow f_i^m$. However, once recovery occurs, and ck_i^4 is restored, e_i^4 and f_i^m become members of E_lost and disappear from the current event set. Therefore, $e_i^4 \not\Rightarrow f_i^m$ when recovery is complete.

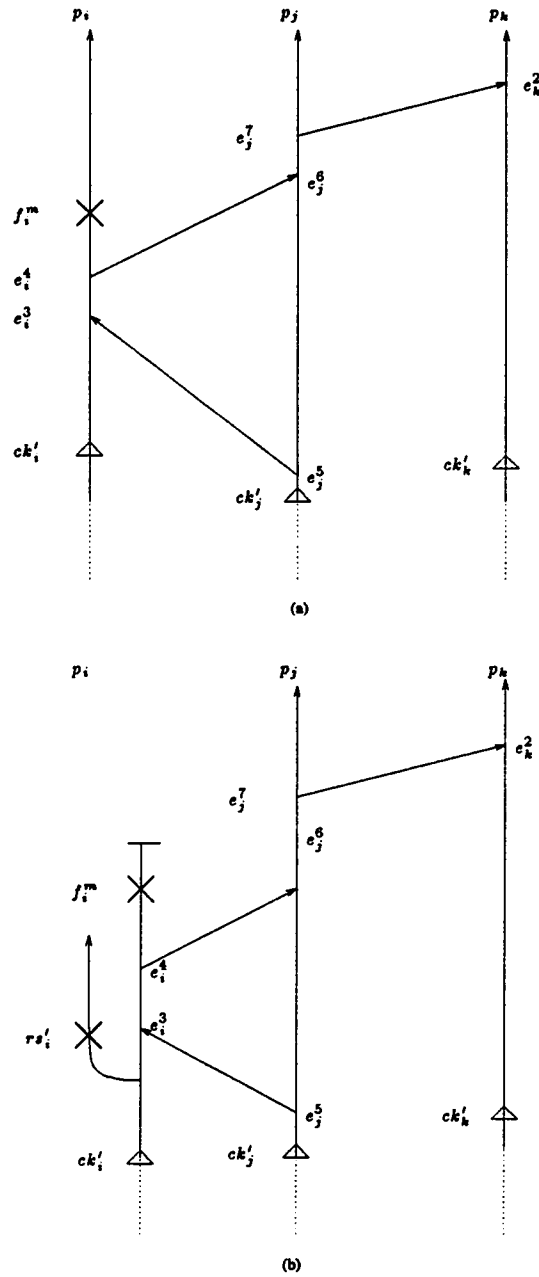


Figure 4.8: Persistence of Causal Effect of Message Transmission

Lost events in the failed process have no causal relationship to the restart event. In Figure 4.8 neither $e_i^4 \Rightarrow rs_i^m$, nor $rs_i^m \Rightarrow e_i^4$. Using the standard definition of causality,

$rs_i^m \not\Rightarrow e'_j \wedge e'_j \not\Rightarrow rs_i^m$, implies $e'_j \parallel rs_i^m$. However, describing these events as concurrent obscures their actual relationship. In actuality when events are lost due to failure or rollback, they are disconnected from the partial order and the events which remain in $E_{current}$. Therefore, we define the following predicate which will allow us to identify these events which have been eliminated from the causal order by process failure or rollback.

$$Disc(e''_i, e'_i) \text{ iff } e''_i \not\Rightarrow e'_i \wedge e'_i \not\Rightarrow e''_i, e''_i \neq e'_i$$

Using this specification for $Disc()$ we can define for a given failure event, a predicate $Orphan$, on the set of system events as follows:

$$Orphan(e'_i, f_k^m) = \begin{cases} \text{True} & \text{if } \exists e'_k \text{ such that } Disc(e'_k, rs_k^m) \wedge e'_k \Rightarrow e'_i, \\ \text{False} & \text{otherwise.} \end{cases}$$

The events which need to be eliminated (removed from $E_{current}$) during rollback are those for which $Orphan()$ is true. The problem is then reduced to identifying which events in each process satisfy the $Orphan()$ predicate.

From the definition of $Orphan()$ it is clear that the causal relationship between events can be used to determine which processors must be rolled back. We will use vector clocks to develop our causal protocol. Consequently, each process will maintain vector clocks and tag its transmissions with vector timestamps. Vector clocks as defined above were not designed to accommodate the loss of events from process execution history. When the system is in an inconsistent state, the isomorphism between the partial order imposed by \rightarrow and the order of vector clock values breaks down. Figure 4.9 shows a system before failure and

after recovery. Events e_j^2 , e_j^3 , and e_k^1 are orphaned by f_i^m . The vector timestamps of the orphan events imply that rs_i' “happens before” them, when in fact rs_i' is disconnected to the orphan events. This anomaly can be used to identify orphan events that must be eliminated to return the system to a consistent state. Returning the system to a consistent state re-establishes the isomorphism between vector clock values and the causal partial order so that if $\neg Orphan(e_i', f_i^m)$ and $\neg Orphan(e_j', f_i^m)$ then

1. $e_i' \Rightarrow e_j'$ iff $V_i(e_i') < V_j(e_j')$
2. $e_i' \parallel e_j'$ iff $V_i(e_i') \parallel V_j(e_j')$.

4.3.3 Correctness Specifications and Polling Waves

When a failed process restarts it retrieves its latest checkpoint from stable storage. The message log is replayed until it is exhausted to restore as much of the failed process' state as possible. The replaying of send events affects only the failed process' state. No duplicate messages are sent as a result of replay. Once the logged messages have been recovered the recovering process instigates a restart event, rs_i^m and begins the rollback protocol.

At this point the recovering process must communicate with all other processes. Actually only those processes which have orphaned events need to be contacted. In practice, however, it is difficult to determine which processes these are without contacting every process. There are several techniques which can be used to contact all the processes. A virtual ring of processes can be derived and a token message circulated, as in Dijkstra's termination detection protocol [15]. A spanning tree with a designated process at the root may also be used [14]. It is also possible for the recovery process to broadcast recovery information to all other processes in the system to begin the rollback procedure [44, 47].

The common theme, in each of these techniques, is that every process must be contacted at least once to indicate that failure has occurred and to send it information necessary for recovery. We characterize this process as a series of one or more *polling waves*. A

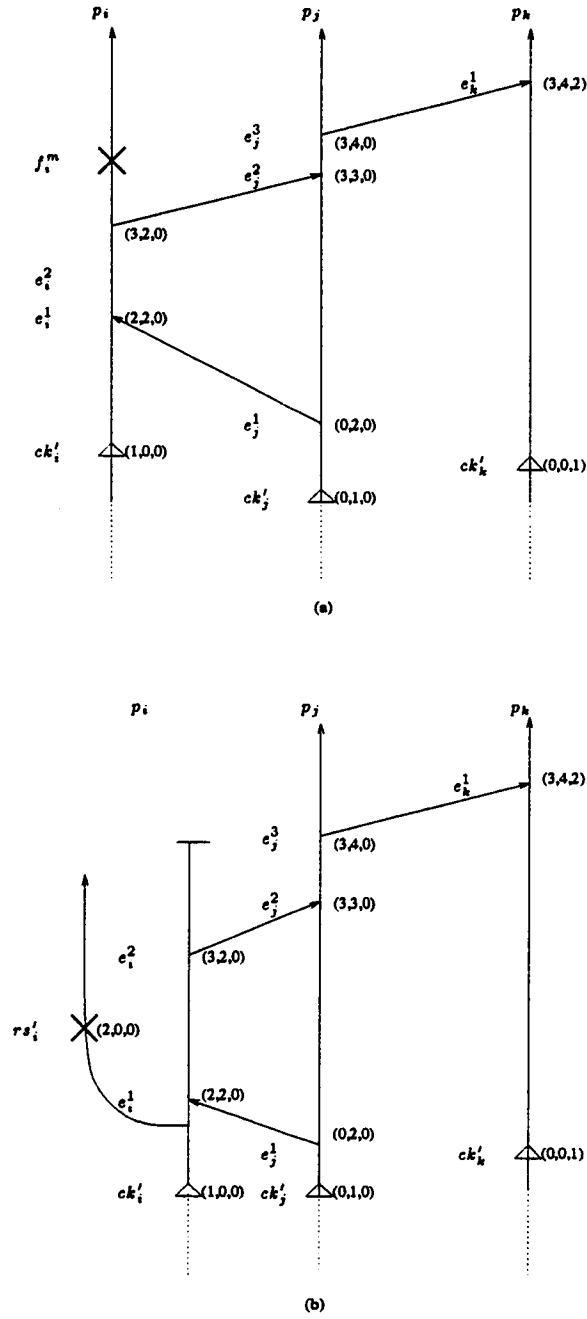


Figure 4.9: Vector Time of Orphan Events

polling wave is characterized by the arrival of a polling message which transmits information

necessary for rollback and some response by the polled process which is also integral to the algorithm. There will necessarily be some last contact with each process. To that end we define two new event types:

- $c_k(i, m)$: the arrival of the final polling wave message for rollback from failure f_i^m at process p_k .
- $w_k(i, m)$: the response to this final polling wave. If no response is required, $w_k(i, m) = c_k(i, m)$.

The final polling wave is denoted

$$FW(i, m) = \bigcup_{k=0}^{N-1} w_k(i, m) \cup \bigcup_{k=0}^{N-1} c_k(i, m).$$

Using this generic model we can define what it means for a rollback protocol be correct strictly in terms of causality. A protocol that can be described using this polling wave model will insure correct rollback if at the completion of the final wave, $FW(i, m)$, the following conditions hold:

Rollback Conditions - RB

- RB(a)** $\forall w_j(i, m) \in FW(i, m), \neg Orphan(w_j(i, m), f_i^m)$; and
- RB(b)** If $w_{p(s)}(i, m) \Rightarrow \eta(s) \vee \eta(s) \Rightarrow w_{p(s)}(i, m)$ then $\neg Orphan(w_{p(s)}(i, m), f_i^m)$; and
- RB(c)** $\neg Disc(s, w_{\sigma(s)}(i, m))$ if and only if $w_{p(s)}(i, m) \Rightarrow \eta(s) \vee \eta(s) \Rightarrow w_{p(s)}(i, m)$.
-

Intuitively, condition RB(a) insures that every event orphaned by a failure f_i^m is eliminated before the final polling wave is completed. The second condition prevents the acceptance of orphan messages after the polling wave has completed. The third condition, RB(c), requires that any message sent as a result of a lost event is ignored. In addition, RB(c) requires that any message sent during a event which is still a member of the partial order at the completion of the polling wave event is eventually delivered.

The rollback conditions as specified by RB only insure safety conditions. If RB(a), RB(b), and RB(c) are met for a failure f_i^m , then it is not possible for the system to be inconsistent due to this failure. One way to satisfy these conditions is to rollback every process to its initial event. Clearly such a protocol would satisfy RB; however, it is a trivial, and not very efficient, protocol. Further conditions must be specified to design an efficient protocol. Consider the following modified rollback conditions:

Rollback Conditions - RB2

RB2(a) $\forall w_j(i, m) \in FW(i, m), \neg Orphan(w_j(i, m), f_i^m)$; and

RB2(b) $\neg Disc(s, w_{\rho(s)}(i, m)) \wedge \neg Orphan(s, f_i^m)$ if and only if $w_{\rho(s)}(i, m) \Rightarrow \eta(s) \vee \eta(s) \Rightarrow w_{\rho(s)}(i, m)$.

These conditions are stronger than what is needed to specify a correct protocol. The requirement that $\eta(s)$ occur if s is not an orphan event means that any non-orphan event that is lost because of failure or rollback must be restored. This means that the protocol can only eliminate orphan events from the partial order.

The first protocol that we propose satisfies the stronger requirements of RB2. It eliminates an event during rollback only if it is an orphaned event. It also restores to the partial order any message receipts that originate in non-orphaned send events. One of the assumptions that makes this protocol possible is that there are no concurrent failures in the system. In Section 4.7 we will eliminate this assumption and show how our protocol can be modified to handle concurrent process failure. The modified protocol satisfies RB, but not RB2.

The polling wave model and rollback conditions based on causality provide a different framework for judging correctness of optimistic recovery protocols than what is commonly

used. Normally correctness of optimistic recovery is evaluated in terms of the execution history of the *environment*, i.e., the external behavior of the distributed computation. The environment of a distributed computation is a special process in the sense that this is where the committed outputs are made. A correct optimistic recovery protocol will guarantee that the execution history of the environment of a computation subject to the recovery protocol is equivalent to a possible failure-free execution history.

The rollback conditions that we have developed do not directly address this issue. However, they can be readily used to argue that any protocol that satisfies them will also guarantee that the execution history of the environment is equivalent to some failure-free execution history. To see this consider the environment as the N^{th} process. The polling wave would not pass through the environment. However, a correct rollback-recovery protocol should be structured so that the conditions specified by RB or RB2 hold for $\Pi \cup \{p_N\}$. If RB holds for $\Pi \cup \{p_N\}$, then there are no orphan events in p_N preceding or following the wave, and every message that originates in a send event left in place by the protocol will eventually be delivered. Therefore, the execution history in the environment p_N must correspond to some failure-free execution. The conditions specified by RB2 also guarantee no orphan events in p_N . Additionally, they require that no non-orphaned events be eliminated due to rollback. The conditions specified by RB and RB2 are satisfied for internal processes through rollback. In most cases the environment cannot be rolled back. Therefore, the only way to guarantee that the Rollback Conditions are met for the environment is to control communication to the environment through a commitment protocol. If commitment of outputs to the environment is done correctly, we can insure that RB or RB2 is satisfied and none of these committed outputs are ever rolled back, thus satisfying the requirement that the external actions of the protocol match a possible failure-free execution. Our commitment method is described in Section 4.4.5.

The first protocol we present uses a single polling wave and vector clocks to perform rollback and recovery using $O(N)$ messages per failure. It is similar to Strom and Yemini's

protocol in that it augments vector timestamps with a per-process incarnation number to distinguish between orphan and non-orphaned states. This protocol illustrates how a firm understanding of causality's roll in this problem can be used to simplify and formalize the protocol specification and correctness arguments.

The protocol has two weaknesses. As in Sistla and Welch's protocol it will not tolerate multiple, concurrent failures. The protocol could be altered to accommodate such failures but it would complicate the protocol significantly. This is a consequence of its second weakness, which is the use of incarnation numbers to restore the isomorphism between the ordering of the vector clock values and the causal partial order that is destroyed by failure and rollback. In a later section we will present protocols which do not use incarnation numbers, and rely strictly on vector clocks and the causal relationships imposed by the polling wave. This protocol is valuable in the sense that it is a straightforward $O(N)$ algorithm that is more efficient than those in [44, 47].

4.4 Causal Recovery Protocol: Single Wave - Serial Failure

4.4.1 Informal Description

Each send and receive event in the application computation increments vector time. The current vector clock value of a process is considered part of its state and is logged to stable storage when a checkpoint is done. $V_i(p_i)$ will signify the current vector clock value of process p_i where e'_i is the most recent event in p_i , and $V_i(p_i) = V_i(e'_i)$. Each time a checkpoint is taken in a process, or a restart event occurs, the vector clock value of that process is incremented. Checkpoints and restart events are the only events of the rollback protocol that cause vector time to advance. None of the polling events of the rollback procedure cause vector time to be incremented.

Each process also maintains a current incarnation number, Inc_i and a vector of sequence numbers, $V.seq_i$, as part of its state. The process incarnation number is updated during

the rollback protocol. The vector of sequence numbers is updated whenever a message is received, so that, $V.seq_i(j)$ equals $V_i^j(\eta(s))$, where $\sigma(s) = j$.

When a failed process restarts, it retrieves its latest checkpoint, including its vector clock value, from stable storage. This value is $V_i(Latest.ck(f_i^m))$, the vector clock value of the last checkpoint taken before failure f_i^m . The message log is replayed until it is exhausted. The vector time of each message is logged with the message so as the messages are replayed the clock value of the failed process can be appropriately updated. After the logged messages have been recovered the recovering process instigates a restart event, rs_i^m , to begin the rollback protocol and then originates a *token* message containing the vector timestamp of rs_i^m . The token associated with failure f_i^m and restart event rs_i^m is designated $tk(i, m)$.

The token is composed of four fields:

- $tk(i, m).ts = V_i(rs_i^m)$
- $tk(i, m).id = i$
- $tk(i, m).inc = Inc_i$
- $tk(i, m).seq = V.seq_i$

Process p_i buffers all incoming messages until the return of the token. When this occurs p_i resumes normal execution.

The token is circulated through the ring of processors. When the token arrives at process p_j the event $c_j(i, m)$ occurs. The timestamp in the token is used to determine whether the process must be rolled back. If $tk(i, m).ts < V_j(p_j)$ then an orphaned event has occurred in p_j , and p_j must be rolled back to an earlier state. This is accomplished by instantiating p_j to the state of ck'_j , where ck'_j is the latest checkpoint for which $V_j(ck'_j) < tk(i, m).ts$, and then replaying logged messages as long as the timestamps of the messages are less than $tk(i, m).ts$.

It is possible that an orphan event in p_j is the receipt of a message originating in a non-orphaned send event. Since the send event corresponding to such a receipt is not an orphan, it does not causally succeed any lost event in p_i . Therefore, the recovery of p_i will not result in the replay of these messages. To make sure that these messages are not lost, p_j must request their retransmission during the rollback step.

During rollback, p_j must also retransmit any message that it sent to p_i that was lost due to failure. Process p_j can determine whether the messages it has sent have been received by the failed process p_i by comparing the vector timestamps of these messages to $tk(i, m).seq$. If $V_j^j(s) > tk(i, m).seq(i)$, where s is a message that was sent to p_i , then it is possible the failed process has lost the message and it must be resent. Because it is also possible that the message is not actually lost, but is still in transit, p_i must discard any duplicate messages. Because channels are FIFO, p_i can identify any duplicate message from its timestamp.

After the logged messages have been replayed and required message retransmissions are done, p_j instigates a rollback event, rb_j^k , to indicate that rollback is complete. Vector time is not incremented for this event, so $V_j(rb_j^k) = V_j(e_j')$, where e_j' is the last event replayed. Any logged event whose vector time exceeds $tk(i, m).ts$ is discarded. In the case where $tk(i, m).ts \not\leq V_j(p_j)$, when the token arrives the state of p_j is not changed. However, for consistency, a rollback event is instigated to indicate that rollback is complete at p_j and to allow the token to be propagated.

When rollback is completed, $V_j(p_j) \not\leq V_i(rs_i^m)$. In causal terms this means that every event in p_j either happens before or is concurrent to any lost event in p_i . When we present our correctness proof we will show how the properties of vector time can be used to prove this.

The token is propagated from p_j to $p_{j+1(mod N)}$, eliminating orphan events as it goes, until it returns to the originating process p_i . When this occurs rollback is complete.

There is one complication in this procedure. It is possible for orphaned messages to be in transit during the rollback process. If these messages are received and processed during

or after the rollback procedure, an inconsistent global state will result. Rollback condition RB2(b) is specified to prohibit this occurrence. To identify these orphan messages and discard them on arrival, it is necessary to include an *incarnation number* with each message and with the token. Inc_i equals the current incarnation number of the process p_i . The function $Inc(e'_i)$ denotes the incarnation number of event e'_i . The value returned for an event equals the current incarnation number of the process in which the event occurred. The incarnation number in the token is designated by $tk(i, m).inc$.

When p_i initiates the rollback process it increments its current incarnation number by one and attaches it to the token. A process receiving the token must save both the vector timestamp of the token and the incarnation number in stable storage. Because there is no bound on message transmission time, the vector timestamps and associated incarnation numbers which have arrived in the token must be accumulated in a set. The set $OrVect_i$ is composed of ordered pairs of token timestamps and incarnation numbers received by p_i . We describe practical techniques for bounding the size of $OrVect_i$ at the end of this section.

When an application message is received by process p_i , the vector timestamp of the message is compared to the vector timestamps stored in $OrVect_i$. If the clock value of the message is found to be greater than a timestamp in $OrVect_i$ then the incarnation number of the message is compared to the incarnation number corresponding to the timestamp in $OrVect_i$. If the message incarnation number is the lesser of the two, then the message is discarded. Clearly this is an orphaned message that was in transit during the rollback process. In all other cases the message is accepted. Upon receipt of a token the receiving process sets its incarnation number to that in the token.

4.4.2 Formal Specification

Retransmission of messages lost by the failed process and requests for retransmission of messages lost during rollback are instigated by the rollback-recovery protocol and are not part of the underlying computation. The following formalizes the rules followed by the

protocol during rollback to insure the necessary retransmissions are accomplished:

Retransmit(tk.ts, tk.id, tk.seq, id, c.event)

For all $e'_{id} = \eta(s)$ such that:

$$\begin{aligned} \eta(s) &\Rightarrow c.event \wedge \\ V_{id}(\eta(s)) &> tk.ts \wedge \\ V_{\sigma(s)}(s) &\not> tk.ts \end{aligned}$$

retransmission of message from $p_{\sigma(s)}$ is requested

For all $e'_{id} = s$ such that:

$$\begin{aligned} s &\Rightarrow c.event \wedge \\ V_i^i(s) &> tk.seq(i) \wedge \\ V_{\sigma(s)} &\not> tk.ts \end{aligned}$$

s is retransmitted to $p_{tk.id}$

The following rules specify the protocol. In each case we formally describe the rule in terms of events and then, in the italicized text, describe verbally the impact of the rule. The formal specifications are used to make a formal argument for the validity of the protocol in as concrete terms as possible.

Causal Recovery Protocol : Single Wave - Serial Failure**Initial Conditions**

$$\begin{aligned}
Inc_i &= 0 \wedge \\
V_i^j(p_i) &= 0 \text{ for all } p_i \in \Pi, j = 0, 1, \dots, N - 1
\end{aligned}$$

CRB.1 The occurrence of rs_i^m implies

$$\begin{aligned}
LastEvent(f_i^m) &\Rightarrow rs_i^m \wedge \\
tk(i, m).ts &= V_i(rs_i^m) \wedge \\
tk(i, m).id &= i \wedge \\
tk(i, m).inc &= Inc(Latest.ck(f_i^m)) + 1 \wedge \\
Inc_i &= Inc(Latest.ck(f_i^m)) + 1.
\end{aligned}$$

A restart event occurs when the latest event that occurred prior to failure is recovered from stable storage. A token incorporating the timestamp of the restart event, the id of the recovering process, and the current incarnation number is created during this event.

CRB.2. $w_i(i, m)$ occurs iff

$$\begin{aligned}
&\exists rs_i^m \text{ such that } rs_i^m \Rightarrow w_i(i, m) \wedge \\
&\exists ck_i' \text{ such that } ck_i' \Rightarrow w_i(i, m) \wedge CK(ck_i', rs_i') \wedge \\
&\left[\begin{array}{l} \exists e_i' \text{ such that } rs_i^m \Rightarrow e_i' \Rightarrow w_i(i, m) \wedge \\ e_i' \text{ is an event of the underlying computation} \end{array} \right]
\end{aligned}$$

A formerly failed process creates and propagates a token, event $w_i(i, m)$, only after the occurrence of a restart event rs_i^m .

CRB.3. The occurrence of an rb'_j event instigated by rs_i^m implies

$$\begin{aligned}
& c_j(i, m) \rightarrow rb'_j \wedge \\
& e'_j \Rightarrow rb'_j \text{ iff } V_j(e'_j) \not\geq tk(i, m).ts \wedge \\
& \text{Retransmit}(tk(i, m).ts, tk(i, m).id, tk(i, m).seq, j, c_j(i, m)) \wedge \\
& Inc_j = tk(i, m).inc \wedge \\
& (tk(i, m).ts, tk(i, m).inc) \in OrVect_j.
\end{aligned}$$

A rollback event will occur in a non-failed process only after all events with timestamps greater than the token have been eliminated and the necessary retransmissions have been requested. The process must have also incremented its incarnation number, and stored the vector timestamp of the token and the incarnation number of the token in its OrVect set.

CRB.4. $w_j(i, m), i \neq j$ occurs iff

$$\begin{aligned}
& \exists rb'_j \text{ instigated by } rs_i^m \text{ such that } rb'_j \Rightarrow w_j(i, m) \wedge \\
& \exists ck'_j \text{ such that } ck'_j \Rightarrow w_j(i, m) \wedge CK(ck'_j, rb'_j) \wedge \\
& \left[\begin{array}{l} \exists e'_j \text{ such that } rb'_j \Rightarrow e'_j \Rightarrow w_j(i, m) \wedge \\ e'_j \text{ is an event of the underlying computation.} \end{array} \right]
\end{aligned}$$

A non-failed process will propagate the token only after it has rolled back and checkpointed the process state at the rollback event.

CRB.5. The occurrence of $\eta(s)$ where $\rho(s) = p_i$, and $rs_i^m \Rightarrow \eta(s)$, implies that $c_i(i, m) \Rightarrow \eta(s)$.

A recovering process will not accept any incoming messages until the first polling wave is completed.

CRB.6. Polling wave $PW(i, m)$ is complete when $c_i(i, m)$ occurs.

When the process which failed, recovered, and initiated the token receives its token, the rollback is complete.

CRB.7. Any message received by event, $\eta(s)$, is discarded iff $\exists m \in OrVect_{\rho(s)}$ such that

$$Inc(s) < Inc(m) \wedge V(m) < V(s).$$

Messages which were in transit and which were orphaned by the failure and subsequent restart and recovery must be discarded.

4.4.3 An Example

In Figure 4.10 we see a system of three processes. The processes take checkpoints at ck_0^1 , ck_1^1 , and ck_2^1 . Each event on a process time line is tagged with the vector time of its occurrence. Each message is tagged with $[i](x, y, z)$, where i is the incarnation number associated with the message send event, and (x, y, z) is the vector time of the send. Process p_0 fails just after the message receipt which increments its vector clock to $(5, 5, 0)$.

The execution of the causal recovery protocol is shown in Figure 4.11. Upon restart of p_0 , the checkpoint ck_0^1 is restored, and the restart event, rs_0^1 is performed by the protocol. A token, appearing as $[1](4, 0, 0)$ is created and propagated to p_1 (the dashed lines indicate token transmission). Upon receipt of the token, p_1 rolls back to a point meeting the requirement that its vector time is not greater than $(4, 0, 0)$, the time in the token. Hence p_1 rolls back to its state at time $(1, 3, 0)$. p_1 then records the token in its *OrVect* set. Finally, it sends the token to p_2 . p_2 takes action similar to p_1 to roll back to time $(1, 3, 2)$. The token is then returned to p_0 , and recovery is complete.

Two messages are in transit while the polling wave is taking place. The message from

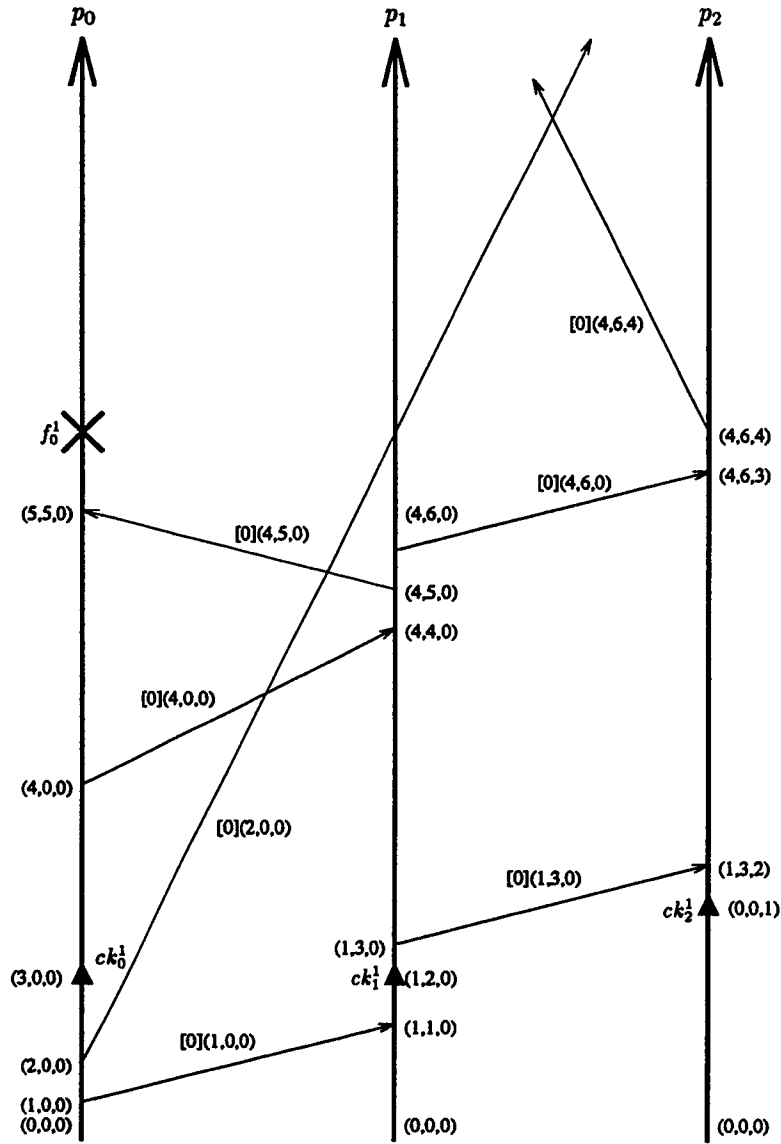


Figure 4.10: Causal Protocol - Single Wave

p_0 to p_2 with label $[0](2,0,0)$ will be accepted when it arrives. Application of Rule CRB.7 will result in message $[0](4,6,4)$ being discarded when it arrives at p_1 .

The net effect of the recovery process is that the application is rolled back to the consistent global state indicated by the bold line, and all constituent processes have sufficient

information to discard messages sent from orphaned events on their arrival.

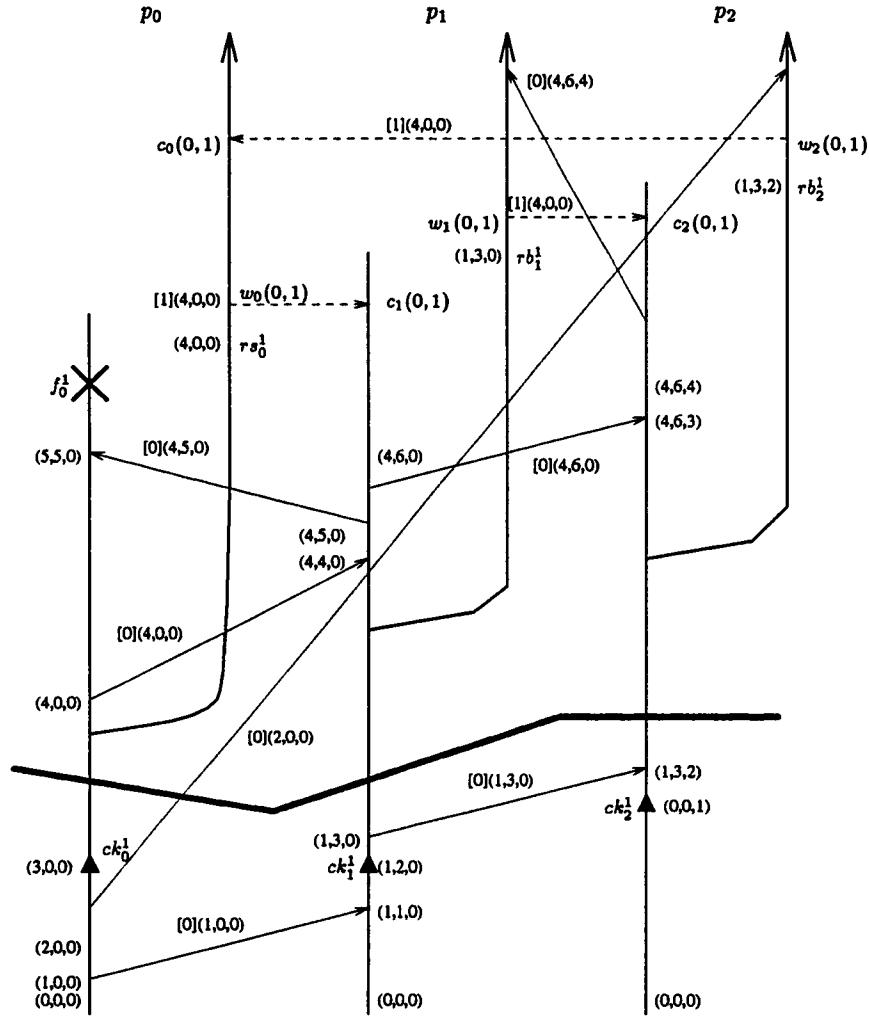


Figure 4.11: Causal Protocol - Single Wave

4.4.4 Correctness

When no restrictions are placed on the occurrence of failure in the system, it is possible for messages and states that are part of the recovery protocol to be lost. In this section we limit failure during the recovery process. This simplifies the problem and the protocol.

In this less complex environment we will prove the validity of our method. In Section 4.7 we will eliminate these restrictions on failure and present a modified protocol which will accommodate concurrent failure.

This protocol is resilient to multiple failures if they occur serially. Two failures are defined to have occurred serially if all the polling events instigated by one failure complete before another failure occurs. The formal conditions for serial failure are stated below. The first disjunct specifies that any event in a polling wave of one failure must occur before any failure that might occur in the same process as the event. The second disjunct requires that no event of one polling wave be orphaned by another failure. The final two disjuncts specify that polling waves from two failures may not overlap one another. Formally, two failures, f_i^m and f_j^k , occur serially if and only if:

$$\begin{aligned} e'_i &\Rightarrow f_i^m, \text{ for all } e'_i \in FW(j, k) \wedge \\ \neg Orphan(e'_x, f_i^m), &\text{ for all } e'_x \in FW(j, k) \wedge \\ w_l(j, k) &\Rightarrow c_l(i, m), \text{ for all } p_l \in \Pi \wedge \\ c_j(j, k) &\Rightarrow c_j(i, m), \end{aligned}$$

or

$$\begin{aligned} e'_j &\Rightarrow f_j^k, \text{ for all } e'_j \in FW(i, m) \wedge \\ \neg Orphan(e'_x, f_j^k), &\text{ for all } e'_x \in FW(i, m) \wedge \\ w_l(i, m) &\Rightarrow c_l(j, k), \text{ for all } p_l \in \Pi \wedge \\ c_i(i, m) &\Rightarrow c_i(j, k). \end{aligned}$$

Our first result establishes the fact that the token, as constructed during the restoration of a formerly failed process, contains the information necessary to determine if any event is orphaned by a failure.

Lemma 36 $\forall e'_i$ such that $\text{Disc}(e_i, rs_i^m)$, $V_i(rs_i^m) \leq V_i(e'_i)$.

Proof: If $\text{Disc}(e'_i, rs_i^m)$ then $\text{LastEvent}(f_i^m) \Rightarrow e'_i$ in the failed execution history, and $V_i(\text{LastEvent}(f_i^m)) < V_i(e'_i)$. This implies $V_i^i(\text{LastEvent}(f_i^m)) < V_i^i(e'_i)$, and for all $j \neq i$, $V_i^j(\text{LastEvent}(f_i^m)) \leq V_i^j(e'_i)$. The vector clock value of rs_i^m differs from $V_i(\text{LastEvent}(f_i^m))$ only in the i^{th} position, and thus, $V_i^i(rs_i^m) = V_i^i(\text{LastEvent}(f_i^m)) + 1$. Therefore, $V_i^i(rs_i^m) \leq V_i^i(e'_i)$, and $V_i(rs_i^m) \leq V_i(e'_i)$ ■

Lemma 37 shows that every orphaned event has a timestamp greater than the token timestamp. The converse is not always true. Lemma 38 proves the converse for events that happen concurrently to $c_i(i, m)$.

Lemma 37 If $\text{Orphan}(e'_j, f_i^m)$ then $tk(i, m).ts < V_j(e'_j)$.

Proof: By the hypothesis, $\text{Orphan}(e'_j, f_i^m)$. Then there exists e'_i such that $\text{Disc}(e'_i, rs_i^m)$, and $e'_i \Rightarrow e'_j$. By Lemma 36, $\text{Disc}(e'_i, rs_i^m)$ implies $V_i(rs_i^m) \leq V_i(e'_i)$. Therefore, $V_i(rs_i^m) \leq V_i(e'_i) < V_j(e'_j)$. Because $tk(i, m).ts = V_i(rs_i^m)$ (CRB.1), $tk(i, m).ts < V_j(e'_j)$. ■

Lemma 38 $\forall e'_j$ such that $c_i(i, m) \not\Rightarrow e'_j$, if $tk(i, m).ts < V_j(e'_j)$ then $\text{Orphan}(e'_j, f_i^m)$.

Proof: Suppose that $tk(i, m).ts < V_j(e'_j)$, and $c_i(i, m) \not\Rightarrow e'_j$, for some event e'_j . This implies that $V_i(rs_i^m) < V_j(e'_j)$ (by CRB.1). This in turn implies that there must exist at least one event e_i^k such that $e_i^k \Rightarrow e'_j$. Let e_i^k be the latest of the events in p_i such that $e_i^k \Rightarrow e'_j$. If this is the case, then $V_i^i(e_i^k) = V_j^j(e'_j)$. The facts that $V_i(rs_i^m) < V_j(e'_j)$, and $V_i^i(e_i^k) = V_j^j(e'_j)$ imply that $V_i(rs_i^m) \leq V_i(e_i^k)$. Therefore, $e_i^k \not\Rightarrow rs_i^m$.

Rule CRB.5 specifies that no receive event, and therefore, no send event occurs between rs_i^m , and $c_i(i, m)$. Since e_i^k is the latest event in p_i such that $e_i^k \Rightarrow e'_j$, e_i^k must be a send event, and $rs_i^m \Rightarrow c_i(i, m) \Rightarrow e_i^k$. However, this implies $c_i(i, m) \Rightarrow e'_j$ contradicting the hypothesis. Therefore, $rs_i^m \not\Rightarrow e_i^k$, $e_i^k \not\Rightarrow rs_i^m$, and $\text{Disc}(e_i^k, rs_i^m)$. By definition, if $\text{Disc}(e_i^k, rs_i^m)$, and $e_i^k \Rightarrow e'_j$, then $\text{Orphan}(e'_j, f_i^m)$. ■

Having established the preliminary result of Lemmas 37 and 38, we proceed to show that the Causal Recovery Protocol satisfies the first rollback requirement: that all orphan events are detected and eliminated.

Lemma 39 *For any $w_j(i, m)$ event as it is specified in the Causal Recovery Protocol, $\neg Orphan(w_j(i, m), f_i^m)$.*

Proof: Assume the contrary, a polling event, $w_j(i, m)$, exists for which $Orphan(w_j(i, m), f_i^m)$. If that is the case, then there exists e'_i such that $Disc(e'_i, rs_i^m)$, and $e'_i \Rightarrow w_j(i, m)$. Then there must exist e'_j such that $e'_i \Rightarrow e'_j \Rightarrow w_j(i, m)$. This implies $Orphan(e'_j, f_i^m)$, and by Lemma 37, $tk(i, m).ts < V_j(e'_j)$. This contradicts Rule CRB.3 of the rollback protocol. ■

The following lemmas establish that non-orphaned messages are delivered and that orphaned messages in transit during or after a failure and recovery are discarded.

Lemma 40 *If $\neg Orphan(s, f_i^m) \wedge \neg Disc(s, rs_i^m)$ then $\eta(s) \Rightarrow w_{\rho(s)}(i, m) \vee w_{\rho(s)}(i, m) \Rightarrow \eta(s)$.*

Proof: $\neg Disc(s, w_{\sigma(s)}(i, m))$ implies $s \Rightarrow w_{\sigma(s)}(i, m)$, or $w_{\sigma(s)}(i, m) \Rightarrow s$. $s \Rightarrow w_{\sigma(s)}(i, m)$ implies $tk(i, m).ts \not\prec V_{\sigma(s)}(s)$, and $\neg Orphan(s, f_i^m)$. By Lemma 39, $w_{\sigma(s)}(i, m) \Rightarrow s$ also implies $\neg Orphan(s, f_i^m)$.

Let s be a send such that $\neg Orphan(s, f_i^m)$. Given reliable channels the message will eventually arrive. The receipt of the message can only disappear from the causal order if it is lost by a failed process, rolled back by the protocol, or discarded upon arrival. The first possibility is that p_i (the failed process) lost the message due to its failure. Note that in this case $\rho(s) = i$. During the rollback at $p_{\sigma(s)}$, this message will be retransmitted. The occurrence of the rb event associated with $w_{\sigma(s)}(i, m)$ guarantees this because $V_{\sigma(s)}^{\sigma(s)} > tk(i, m).seq(\sigma(s))$ (Rules CRB.3 and CRB.4). Therefore, $w_i(i, m) \Rightarrow \eta(s)$. The second possibility is that $\eta(s) \Rightarrow c_{\rho(s)}(i, m)$, and $\eta(s)$ was rolled back because $Orphan(\eta(s), f_i^m)$.

However, $V_{\sigma(s)}(s) \not\leq tk(i, m).ts$. Therefore, $p_{\rho(s)}$ will request retransmission before the occurrence of the rb event, and $w_{\rho(s)}(i, m) \Rightarrow \eta(s)$ (Rules CRB.3 and CRB.4).

The final possibility is that $\eta(s)$ occurs after the wave but is discarded upon arrival. By Rule CRB.7, $\eta(s)$ is discarded if and only if $V_{\sigma(s)}(s) > tk(i, m).ts$, and $Inc(s) < tk(i, m).inc$. If $s \Rightarrow w_{\sigma(s)}(i, m)$ then $V_{\sigma(s)}(s) \not\leq tk(i, m).ts$. Therefore, the message will be accepted, and $w_{\rho(s)}(i, m) \Rightarrow \eta(s)$.

In the case that $w_{\sigma(s)}(i, m) \Rightarrow s$, $Inc(s) \not\leq tk(i, m).inc$ (Rule CRB.3). Therefore, $\eta(s)$ will not be discarded upon arrival, and $w_{\rho(s)}(i, m) \Rightarrow \eta(s)$. It is not possible in this case for $\eta(s)$ to be lost due to p_i 's failure. It is possible that $\eta(s)$ is eliminated through rollback. But in that case $c_i(i, m) \not\leq s$. Therefore, by Lemmas 37 and 38, $V_{\sigma(s)} \not\leq tk(i, m).ts$, and $p_{\sigma(s)}$ will request retransmission of $\eta(s)$. ■

Lemma 41 *If $\eta(s) \Rightarrow w_{\rho(s)}(i, m) \vee w_{\rho(s)}(i, m) \Rightarrow \eta(s)$ then $\neg Orphan(s, f_i^m) \wedge \neg Disc(s, w_{\sigma(s)}(i, m))$.*

Proof: Case 1: Assume $\eta(s) \Rightarrow w_{\rho(s)}(i, m)$. By Lemma 39 $\neg Orphan(w_{\rho(s)}(i, m), f_i^m)$. Therefore $\neg Orphan(\eta(s), f_i^m)$, and $\neg Orphan(s, f_i^m)$. $\eta(s) \Rightarrow w_{\sigma(s)}(i, m)$ implies $tk(i, m).ts \not\leq V_{\sigma(s)}(s)$. Therefore, $s \Rightarrow c_{\sigma(s)}(i, m)$, and $\neg Disc(s, w_{\sigma(s)}(i, m))$, Rule CRB.3.

Case 2: Assume $w_{\rho(s)}(i, m) \Rightarrow \eta(s)$, and $Orphan(s, f_i^m)$. By Lemma 37 this implies that $tk(i, m).ts < V_{\sigma(s)}(s)$. Rules CRB.1 and CRB.2 of the protocol guarantee that if $Orphan(s, f_i^m)$ is true then $Inc(s) < tk(i, m).inc$. Rules CRB.3 and CRB.4 require that $tk(i, m).ts$ and $tk(i, m).inc$ are stored in $OrVect_j$ before $w_j(i, m)$ occurs. Therefore there exists $z \in OrVect_j$ for which $V(z) < V(s)$ and $Inc(z) > Inc(s)$. Rule CRB.7 requires that such a message be discarded contradicting our assumption that $w_{\rho(s)}(i, m) \Rightarrow \eta(s)$. The same argument applies in the case that $Disc(s, w_{\sigma(s)}(i, m))$. ■

Theorem 17 *The completion of a valid wave in the Causal Recovery protocol satisfies RB2(a) and RB2(b).*

Proof: Follows directly from Lemmas 39, 40 41. ■

Theorem 17 shows that RB2 is satisfied at the completion of the polling wave. To finish our correctness arguments, we need to show that the specification of the protocol guarantees the completion of a valid wave, so that, $rs_i^m \rightsquigarrow c_i(i, m)$. Before we can do this we must talk about what progress means given failure.

In a failure free system, once an event occurs it remains part of the execution history. To show $e'_i \rightsquigarrow e'_j$, it is sufficient to argue that if e'_i occurs then eventually e'_j occurs. Underlying arguments about progress is the implicit assumption that events are stable in some sense. Once an event occurs it doesn't disappear. In a failure prone system this assumption doesn't hold. It is not enough to show, for example, that $c_j(i, m)$ leads to $w_j(i, m)$ because the failure of the p_j at some time in the future can result in the loss of $w_j(i, m)$. To show progress we must not only show that the protocol guarantees the occurrence of an event, but we must show that the event will never be lost from the current set of events. An event that cannot be lost from $E_{current}$ is *stable*. We define the following predicate:

$Stable(e'_j)$ iff $\nexists f_i^m$ or rb_j^l instigated by rs_i^m such that $Disc(e'_j, rs_i^m) \vee Disc(e'_j, rb_i^l)$.

Using this predicate we define progress in a failure prone system as follows:

$$\text{If } \left\{ \begin{array}{l} e'_j \text{ occurs} \wedge \\ Stable(e'_j) \wedge \\ e'_i \Rightarrow e'_j \end{array} \right\} \text{ then } e'_i \rightsquigarrow e'_j.$$

To show that $rs_i^m \rightsquigarrow c_i(i, m)$, we first show that all events in the polling waves are stable.

Lemma 42 *If $w_j(i, m) \in FW(i, m)$ then there does not exist f_j^k such that $Disc(e'_j, rs_j^k)$.*

Proof: Assume there exists f_j^k and $w_j(i, m) \in FW(i, m)$ such that $Disc(e'_j, rs_j^k)$. This implies $rs_j^k \not\Rightarrow w_j(i, m)$, and $w_j(i, m) \not\Rightarrow rs_j^k$.

Case 1 ($i \neq j$): Rule CRB.4 specifies that the occurrence of $w_j(i, m)$ implies there exists ck'_j such that $CK(ck'_j, rb'_j)$. The rule also requires that no event of the underlying computation occurs between rb'_j and $w_j(i, m)$. Therefore, $w_j(i, m)$ is always recoverable following failure and $\neg Disc(e'_j, rs_j^k)$.

Case 2 ($i = j$): A similar argument can be made in this case using Rule CRB.2. ■

Lemma 43 *Given a failure, f_i^m , $tk(i, m).inc > Inc(c_l(i, m))$ for all $p_l \in \Pi$.*

Proof: Because failure and recovery are required to be serial in this system environment, we use induction to prove this hypothesis. Let f_i^m be the first failure in the system. So that for all failures, $f_j^k \neq f_i^m$ there exists $w_j(i, m) \in FW(i, m)$ such that $w_j(i, m) \Rightarrow f_j^k$. Initially $Inc_l = 0$ for all $p_l \in \Pi$. Inc_l can only be changed according to rules CRB.1 and CRB.3. Since there exists $w_j(i, m)$ such that $w_j(i, m) \Rightarrow f_j^k$ for any other failures, no process incarnation number can be incremented before the arrival of the token by the application of CRB.1. It is also the case that $w_l(i, m) \Rightarrow c_l(j, k)$ for any $p_l \in \Pi$ and $c_l(j, k) \in FW(j, k)$. Therefore, no process incarnation number can be incremented before the occurrence of $c_l(i, m)$ by the application of CRB.3.

We now assume the hypothesis is true for some failure f_j^k that occurs after the initial failure. So that for all $c_l(j, k) \in FW(j, k)$, $tk(j, k).inc > Inc(c_l(j, k))$. Now we will show that the hypothesis holds for the failure immediately following the completion of recovery from f_j^k . Let f_a^x be this failure, such that there exists $w_a(j, k) \Rightarrow f_a^x$, and there does not exist f_b^y such that $w_a(j, k) \Rightarrow w_a(b, y) \Rightarrow f_a^x$. The application of Rule CRB.3 specifies that $Inc(w_l(j, k)) = tk(j, k).inc$ for all $p_l \in \Pi$ and $w_l(j, k) \in FW(j, k)$. $w_a(j, k) \Rightarrow f_a^x$, so by Rule CRB.1, $tk(a, x).inc > Inc(w_a(j, k))$. Hence, $tk(a, x).inc > Inc(w_l(j, k))$ for all $p_l \in \Pi$ and $w_l(j, k) \in FW(j, k)$. The conditions governing serial failure also specify that $w_l(j, k) \Rightarrow c_l(a, x)$ for all $p_l \in \Pi$. Since there are no intervening failures and recoveries

between f_j^k and f_a^x , $Inc(w_l(j, k)) = Inc(c_l(a, x))$, and $tk(a, x).inc > Inc(c_l(a, x))$ for all $p_l \in \Pi$ ■

Lemma 44 *If $e'_j \in FW(i, m)$ then there does not exist f_a^k or rb'_j such that rb'_j is instigated by rs_a^k , and $Disc(e'_j, rb'_j)$.*

Proof: Assume the contrary, that there exists $w_j(i, m) \in FW(i, m)$, f_a^k , and rb'_j such that $Disc(w_j(i, m), rb'_j)$. Case 1 ($i \neq j$): The conditions of serial failure require that $w_j(i, m) \Rightarrow c_j(a, k)$, or $c_j(a, k) \Rightarrow w_j(i, m)$. First consider the case that $c_j(a, k) \Rightarrow w_j(i, m)$. This implies $w_j(a, k) \Rightarrow w_j(i, m)$, and $rb'_j \Rightarrow w_j(a, k) \Rightarrow w_j(i, m)$. Now consider the possibility that $w_j(i, m) \Rightarrow c_j(a, k)$. In this case $w_j(i, m) \Rightarrow rb'_j$ unless $w_j(i, m)$ is eliminated by the rollback instituted by the occurrence of $c_j(a, k)$. The elimination of $w_j(i, m)$ by rollback implies that the rb event, rb'_j , and checkpoint event specified in Rule CRB.4 must also be eliminated by rollback. In that case, Rule CRB.3 implies $V_j(rb'_j) > tk(a, k).ts$. By Lemma 38 this implies $Orphan(rb'_j, f_a^k)$, and $Orphan(w_j(i, m), f_a^k)$. However, the conditions of serial failure prohibit this. Therefore, $w_j(i, m) \Rightarrow rb'_j$.

Case 2 ($i = j$): A similar argument can be made that $rb'_i \Rightarrow w_i(i, m)$, or $w_i(i, m) \Rightarrow rb'_i$. Rollback of $w_i(i, m)$ implies rollback of rs_i^m . This implies $Orphan(rs_i^m, f_a^k)$ and $Orphan(w_i(i, m), f_a^k)$. This contradicts the conditions imposed by serial failure. ■

Theorem 18 $rs_i^m \rightsquigarrow c_i(i, m)$ in the Causal Recovery Protocol : Single Wave - Serial Failure.

Proof: Lemmas 42 and 44 showed that $Stable(w_j(i, m))$ for all $w_j(i, m) \in FW(i, m)$. According to Rule CRB.2 $w_i(i, m)$ will occur following rs_i^m . Since $w_i(i, m)$ is stable, $rs_i^m \rightsquigarrow w_i(i, m)$. Given reliable communication a token message originating in an event $w_i(i, m)$ will arrive at $p_{i+1(mod N)}$. The restrictions on failure guarantee that the message is not lost, however, Rules CRB.5 and CRB.7 restrict the occurrence of incoming messages. Rule CRB.5 could lead to deadlock if two tokens are traversing the system concurrently.

However, the restrictions of concurrent failure require that $p_{i+1(modN)}$ must have completed any recovery before the arrival of the token. Therefore, the receipt of a polling message would not be blocked by Rule CRB.5. Rule CRB.7 could cause an incoming token to be discarded if its attached incarnation number and vector timestamp do not meet the specified conditions. For the token to be discarded the incarnation number attached to the token must be less than the incarnation number of one of the ordered pairs stored in $OrVect_{i+1(modN)}$. It must be the case that all of these stored incarnation numbers must be less than or equal to $Inc_{i+1(modN)}$. Lemma 43 showed that $tk(i, m).inc > Inc(c_{i+1(modN)}(i, m))$. Since the incarnation number attached to the token message equals $tk(i, m).inc$, it cannot be less than any of the incarnation numbers stored in $OrVect_{i+1(modN)}$, and CRB.7 will not cause the rejection of the token message. Thus, $c_{i+1(modN)}(i, m)$ will occur following $w_j(i, m)$, and $w_i(i, m) \rightsquigarrow c_{i+1(modN)}(i, m)$. Rule CRB.3 implies the occurrence of an rb'_j event following $c_{i+1(modN)}(i, m)$. This in turn implies $w_{i+1(modN)}(i, m)$. Because the token travels in a logical ring $rs_i^m \rightsquigarrow c_i(i, m)$. ■

4.4.5 Commitment

To guarantee that the external behavior of the system is correct, messages cannot be committed to the environment until it is no longer possible that they will be rolled back. A process knows that one of its events will never be rolled back, if all of the messages that causally precede that event are logged in stable storage or otherwise saved in a process state checkpoint and cannot be lost in a failure. Because of the causal dependencies, a process cannot decide whether an event is committable based solely on local information. It must gather information about the state of other processes in the system.

For the purposes of this discussion, an event that is recoverable from stable storage is called a recoverable event. Recoverable events may be discarded due to rollback, but they are never lost because of process failure. A process event that is causally dependent only on recoverable events will never be rolled back. The example in Figure 4.12 illustrates this.

The recoverable events in each process are circled. Events e_0^1 and e_0^2 will never be lost to failure, and neither will they be rolled back because they do not causally depend on any message that could be lost to failure. Similarly, e_2^1 and e_2^2 will never be rolled back because they do not causally depend on any event that might be lost. The set of the latest such events in each process are e_0^2 , e_1^1 , e_2^2 , and e_3^1 . This set of events comprises the consistent cut shown in Figure 4.12 by the dashed line. Any event in this set and all the events that causally precede the events in the set may be committed because they will never be rolled back.

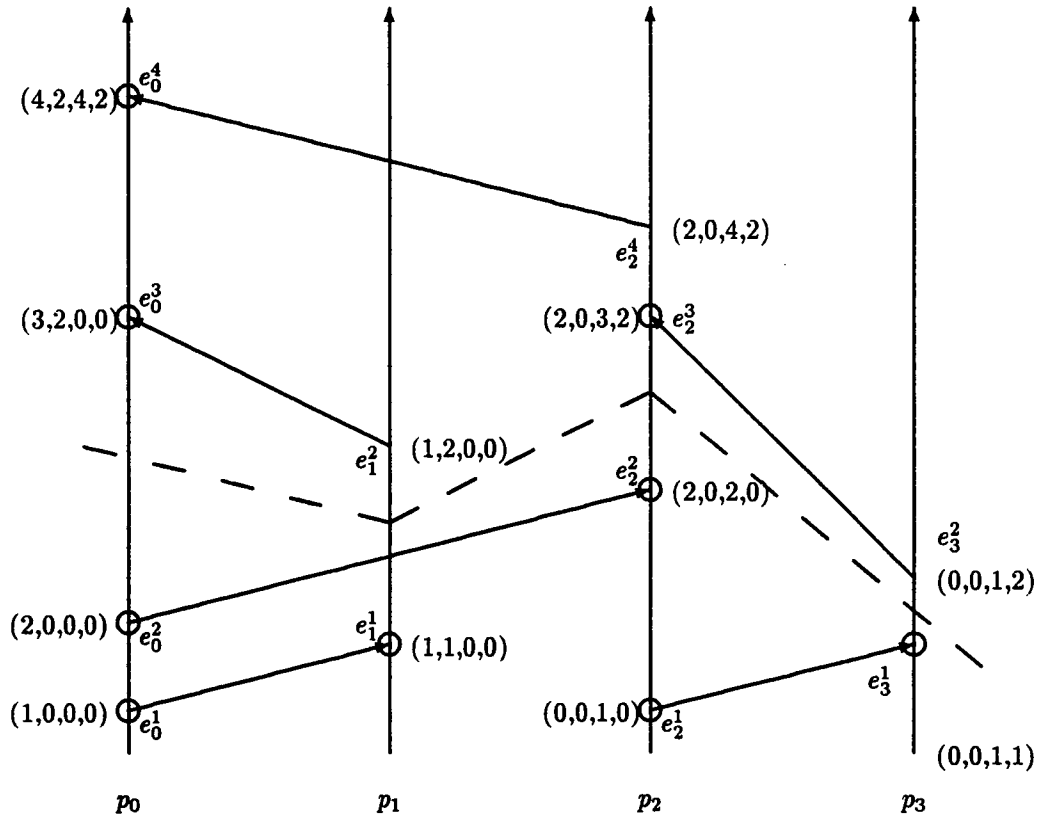


Figure 4.12: Commitment Protocol

A process can determine whether a local event is causally dependent on any event that might be lost by comparing the timestamp of a local event to the timestamps of the latest

recoverable events in the other processes. In this example p_1 can compare $V_1^j(e'_1)$ to $V_j^j(e'_j)$ where e'_j is the latest recoverable event in p_j for all $p_j \neq p_1$ in the system. If $V_1^j(e'_1) \leq V_j^j(e'_j)$ for all $j \neq 1$ then e'_1 will never be rolled back and may be committed to the environment.

A simple means of gathering the latest recoverable event indices is for some process, p_i for example, to circulate a token that builds a composite timestamp from the timestamps of the latest recoverable events in each process. So that when the token, designated by $tk.commit_i$, leaves p_j the j^{th} index of the composite timestamp, $tk.commit_i[j]$, equals $V_j^j(e'_j)$, where e'_j is the latest stable event in p_j . When the token returns to p_i , it has constructed a consistent cut comprised of events that will never be rolled back. With this information, p_i can commit messages to the environment. This information is useful to other processes as well. For this reason, the token is circulated a second time to spread the knowledge of the consistent cut to the other process.

Any process may instigate the token. The criteria used by a process to decide to instigate a token can vary with system requirements, but an obvious trigger would be the accumulation of some predetermined number of messages targeted for the environment.

The following theorem establishes the correctness of this technique.

Theorem 19 *If $V_i^j(e'_i) \leq tk.commit_k[j]$ for some k and all $j \neq i$ then event e'_i will never be rolled back.*

Proof: Assume the converse. Event e'_i can be rolled back only if $Orphan(e'_i, f_k^m)$ for some failure f_k^m . If this is true then $tk(k, m).ts < V_i(e'_i)$ and $tk(k, m).ts[j] \leq V_i^j(e'_i)$ for all j . However, if e'_k is the latest stable event in p_k then $V_k^k(e'_k) < tk(k, m).ts[k] \leq V_i^k(e'_i)$. This contradicts our assumption that $V_i^j(e'_i) \leq tk.commit_k[j]$ for all $j \neq i$. ■

4.4.6 Bounding the Size of *OrVect*

In a system with FIFO channels, elements of *OrVect* can be deleted from the set once a process can determine that all orphan messages associated with a failure have been flushed

from its incoming channels. A process can determine this from the timestamps of incoming messages. When process p_i receives a message from p_j and the j^{th} index of the incoming vector is greater than the j^{th} index of a vector stored in $OrVect_i$, process p_i knows that any orphan message en route from p_j has been flushed from the system. When this is true for all the elements of a vector in $OrVect$, that vector and its associated incarnation number may be deleted.

The protocol presented here is efficient and fairly easy to understand. It uses a token to rollback every process to a consistent state. The resulting state is optimum in that only orphaned events are eliminated. However, it requires incarnation numbers to prevent the system from becoming inconsistent as the result of accepting slow orphan messages. The way rollback is performed here is the cause of this. In previous protocols we have been able to use the causal relationships imposed by the polling wave to deduce properties of the system. In this protocol we don't have that opportunity, because process rollback destroys these causal relationships. It is not true that $w_1(i, m) \Rightarrow w_2(i, m)$, or even that $c_1(i, m) \Rightarrow w_1(i, m)$. This is a result of eliminating the polling events and rolling back time. The synchronous protocol in the next section illustrates how this rolling back of time is unsatisfactory from a logical point of view.

4.5 Real Time and Synchronous Recovery

When developing protocols for termination and deadlock detection we assumed the existence of a global clock and used this assumption to design simple protocols. The availability of global time does not have the clear advantage over vector time in the development of a recovery protocol. The following discussion outlines a synchronous protocol for rollback and recovery that is modelled after the protocol presented in the previous section. It highlights some of the difficulties that occur.

In a system where global real time is available to each processor it is relatively easy

to identify a global state which is consistent with the state of a failed process. All that is necessary is to rollback the non-failed processes to the state they were in at the time of the last recovered event in the failed process. So that if a failed process is able to recover its state at time t_1 , each non-failed process is rolled back so that its state is the same as it was at t_1 . Since no message can be received before it is sent, no process state will include an orphaned event, and a globally consistent state will result. Clearly, a protocol that rolls back every process to a particular global real time satisfies RB2(a). The consistent state that results is less than optimum. The fact that an event occurs later in time than the time of the last recovered event does not necessarily mean the event is an orphan. Therefore, non-orphan events may be rolled back, and this real-time protocol only meets the specifications of RB.

This protocol will use real timestamps rather than vector timestamps to identify the rollback point. In our synchronous protocol, when a failed process restarts, it retrieves its latest checkpoint from stable storage. The time of the checkpoint event, considered part of the processor's state, is saved on stable storage and, therefore, recovered during restart. The message log is replayed until it is exhausted. The recorded time of the last event recovered determines the time to which all the other processes must be rolled back. After the logged messages have been recovered the recovering process instigates a restart event, rs_i^m , to begin the rollback protocol and then originates a *token* message containing the time of the last recovered event. The token associated with failure f_i^m and restart event rs_i^m is designated $tk(i, m)$. The timestamp value of the this token is $tk(i, m).ts$.

Before we can describe the protocol some additional notation is required:

- $T_i(e'_i)$ - local clock value of event e'_i
- $T(e'_i)$ - clock value at time of e'_i occurrence

In the causal protocol the vector time clock of a failed process was restarted at the timestamp value of the restart event, rs_i^m . As processes were rolled back their vector clocks were also set to an earlier value. In a system with a global real time clock it is

not desirable to actually rollback the clock to recover from failure. Instead we will use a correction factor to “logically” turn the clock back. Therefore, if $T(rs_i^m)$ is the clock value of the restart event, and $T_i(\text{LastEvent}(f_i^m))$ is the timestamp of the last recoverable event, $T(rs_i^m) - T_i(\text{LastEvent})$ will be used to simulate the “rollback” of the process clock. Therefore, after all the messages have been replayed the process compares the timestamp of the the last recovered event to the current value of the real time clock T to calculate the correction factor. A checkpoint is taken during the restart event to save the token values and the correction factor. The timestamp of this event and all the following events will be calculated by subtracting this correction factor from the current value of the real time clock. The correction factor must be circulated in the token to keep the clocks of the processes synchronized. The correction factor component of the token is $tk(i, m).cf$.

The token is circulated through a virtual ring of the processors. When the token arrives at a process p_j , the timestamp in the token is used to determine how far the the process must be rolled back. Process p_j must be rolled back so that $T_j(e'_j) \leq tk(i, m).ts$ for all e'_j . This is accomplished by instantiating p_j to the state of ck'_j , where ck'_j is the latest checkpoint for which $T_j(ck'_j) \leq tk(i, m).ts$, and then replaying logged messages as long as the timestamps of the messages are less than or equal to $tk(i, m).ts$. After the logged messages have been replayed p_j instigates a rollback event, rb_j^k , to indicate that rollback is complete. The time of this rollback event must be synchronized with the clock of the recovering process. This is done using the correction factor in the token so that $T_j(rb_j^k) = T(rb_j^k) - tk(i, m).cf$. The timestamp of any event following rb_j^k must also be calculated in this way. Any logged event for which $T_j(e_j)$ exceeds $tk(i, m).ts$ is discarded. Once the events that have later times than the timestamp are discarded and the correction factor has been used to “set” the process clock, the token is propagated to the next process.

As in the previous protocol it is necessary to identify orphan messages that are in transit during the rollback process. This can be done through the use of incarnation numbers as before.

The formal specification and correctness proofs for this protocol follow the same pattern as the ones presented for the first causal protocol, so they will not be presented here. This protocol is inefficient because it may roll back processes farther than necessary. This occurs because the ordering of real timestamps is not isomorphic to the causal partial order in the way that vector timestamps are. The fact that $T_i(e'_i) < T_i(e'_j)$ does not imply $e'_i \Rightarrow e'_j$ the way that $V_i(e'_i) < V_j(e'_j)$ does. Therefore, some events will be eliminated through rollback that are not orphans, but whose timestamps exceed rs_i^m . This means that the protocol satisfies RB, but not RB2. The other undesirable feature to this protocol is that it requires correction factors to rollback time.

This attempt to structure a synchronous protocol after the causal protocol presented in Section 4.4 runs into difficulty because it requires that time go backward. As a consequence, a contrived technique of adding offsets to the real time to construct clock values is required, because it is not reasonable to make real clocks go backward. This is inefficient, counter-intuitive, and, in general, a poor use of perfectly synchronized clocks.

A protocol that has no need for correction factors or incarnation numbers would be preferable. Such a protocol would be dependent solely on event timestamps to perform rollback and ensure that the system remains in a consistent state. In the following synchronous protocol there is no need for time to go “backward”. Timestamps are never reused, therefore, actual clock values are used eliminating the need for correction factors. The resulting protocol seems more natural and less contrived.

As it turns out this also eliminates the need for incarnation numbers. The ordering of the timestamps alone is sufficient to perform rollback and recovery. In Section 4.6 we will show how vector timestamps can be substituted for real timestamps in this protocol to produce a causal protocol that also does not require incarnation numbers to augment vector timestamps. The causal relationships imposed by the polling wave and the ordering of vector timestamps is sufficient when vector time does not go backwards.

4.5.1 Synchronous Rollback - Two Waves

Informal Description

As in the previous protocol a token is circulated through a virtual ring of processors to transmit the rollback and recovery information. Two circuits of the token are required for each failure. The first circuit, i.e., the first polling wave, transmits the information necessary to rollback each process to a consistent state. The second polling wave transmits information needed to prevent the acceptance of messages that originate in orphan events. Arrival and departure events of the token message at each process define the polling waves. We define the following event types.

- $c1_k(i, m)$: the arrival of the first polling wave message for rollback from failure f_i^m at process p_k .
- $w1_k(i, m)$: the response to the first polling wave.
- $c2_k(i, m)$: the arrival of the second polling wave message for rollback from failure f_i^m at process p_k .
- $w2_k(i, m)$: the response to the second polling wave.

The waves of rollback are defined as

$$PW1(i, m) = \bigcup_{k=0}^{N-1} w1_k(i, m) \cup \bigcup_{k=0}^{N-1} c1_k(i, m).$$

and

$$PW2(i, m) = \bigcup_{k=0}^{N-1} w2_k(i, m) \cup \bigcup_{k=0}^{N-1} c2_k(i, m).$$

$$FW(i, m) = PW2(i, m).$$

The current clock value of a process is considered part of its state and is logged to stable storage when a checkpoint is done. The clock values of an incoming message and the receive event are saved with the message content in the message logs.

When a failed process begins recovery it retrieves its latest checkpoint from stable storage and replays messages from the log saved in stable storage. It also recovers the clock value, $T(\text{Latest}.ck(f_i^m))$. Once the message log is exhausted a restart event occurs. The value of $T(rs_i^m)$ will be the current time on p_i 's clock. The timestamp of $\text{LastEvent}(f_i^m)$ will necessarily be less than $T(rs_i^m)$. It is also true that if $T(e'_j) \leq T(\text{LastEvent}(f_i^m))$ then $\neg \text{Orphan}(e'_j)$. It is not possible to say that $T(e'_j) > T(\text{LastEvent}(f_i^m))$ implies $\text{Orphan}(e'_j)$. By comparing the timestamps of every other process state to $T(\text{LastEvent}(f_i^m))$, it is possible to eliminate all orphan events and return the system to a consistent state. This state is the one that existed at $T(\text{LastEvent}(f_i^m))$.

The recovering process generates a token after the occurrence of the restart event. The token is composed of four fields:

- $tk(i, m).ts = T(\text{LastEvent}(f_i^m))$
- $tk(i, m).id = i$
- $tk(i, m).times(j) = T(s)$ where $\eta(s)$ is the latest message received from p_j
- $tk(i, m).Ltime = T(cl_i(i, m))$

The token is circulated through the virtual ring of processors. The arrival of the token at p_j is signified by the event $cl_j(i, m)$. When p_j receives the token, the timestamp of the process' clock before the arrival of the token is compared to the timestamp in the token. So, if e'_j is the latest event before $cl_j(i, m)$, $T(e'_j)$ is compared to $tk(i, m).ts$. If $tk(i, m).ts < T(e'_j)$, event e'_j is possibly an orphaned event, and p_j must be rolled back.

Rollback is accomplished by discarding the current state, recovering the latest checkpoint ck_j^k for which $T(ck_j^k) \not\geq tk(i, m).ts$, and replaying the messages from the log whose timestamps are not greater than the one in the token. After the logged messages have been replayed, all logged events whose timestamps exceed $tk(i, m).ts$ are removed from volatile and stable storage with one exception. Record of the token's arrival is maintained in the logs. When the appropriate events in p_j have been discarded, rollback is complete, and $T(p_j) \not\geq T(rs_i^m)$.

In our first causal protocol, a process p_j , could determine whether p_i had possibly lost a message sent by p_j , by comparing the vector timestamps of such messages to the vector of sequence numbers in the token. In this protocol the clock time of received messages is used to determine which messages must be retransmitted. This necessitates the token field, $tk(i, m).times$, to transmit the timestamp of the latest message received from each process.

During rollback, p_j compares the timestamp of each message sent to p_i (the failed process) to $tk(i, m).times$. If the j^{th} element of the timestamp of any of these messages exceeds the j^{th} element of $tk(i, m).times$, p_i has possibly lost that message. P_j resends any message that it determines has been lost by p_i .

Rolling back a process may eliminate the receipt of messages that originate in events that are not lost due to failure or rollback. These messages will have timestamps less than $tk(i, m).ts$. Process p_j must request retransmission of these messages if RB(c) is to be satisfied. The clock value of each incoming message that will be eliminated during rollback is compared to $tk(i, m).ts$. If the clock value of a message is less than $tk(i, m).ts$, then it originated in a non-lost event. P_j requests retransmission of any messages that meet this criteria. The following specifies the rules followed by the protocol to guarantee that the necessary transmissions are accomplished.

Retransmit(*tk.ts*, *tk.id*, *tk.times*, *id*, *c.event*)

For all $e'_{id} = \eta(s)$ such that:

$$\eta(s) \Rightarrow c.event \wedge$$

$$T(\eta(s)) > tk.ts \wedge$$

$$T(s) \not\leq tk.ts$$

retransmission of message from $p_{\sigma(s)}$ is requested

For all $e'_{id} = s$ such that:

$$s \Rightarrow c.event \wedge$$

$$T(s) > tk.times(i) \wedge$$

$$T(s) < tk.ts$$

s is retransmitted to $p_{tk.id}$

Once rollback is completed, the arrival of $c1_j(i, m)$ is replayed. The state of p_j is updated to include the polling event, and $T(c1_j(i, m))$ is set to the current clock value.

After $c1_j(i, m)$ is replayed, event rb'_j occurs to indicate rollback is complete. A checkpoint of process p_j 's state is taken before the token is propagated. This checkpoint preserves the token information. This information is necessary for future execution of the process, p_j , and must be recoverable following any future failure.

The token is propagated from p_j to $p_{j+1(modN)}$ until it returns to the originating process p_i . When this occurs, the actual rollback portion of the protocol is complete, and every process has been returned to a consistent state. However, it is possible for a message

originating in an orphan send event to be in transit during the rollback process. Our previous protocol used incarnation numbers to prevent such messages from being accepted. In this protocol, the clock value of $c1_i(i, m)$ is used to determine whether a message might have originated in an orphan send event or not. A message sent during an orphan event must have been sent before $T(c1_{\sigma(s)}(i, m))$. Therefore, $T(s) < T(w1_{\sigma(s)}(i, m)) < T(c1_i(i, m))$. It is also the case that if $Disc(s, rs_i^m)$, then $tk(i, m).ts < T(s)$. Using $tk(i, m).ts$ and $T(c1_i(i, m))$, every process can determine whether to accept messages that arrive after the first polling wave.

The second polling wave disseminates $T(c1_i(i, m))$ to all the processes. $T(c1_i(i, m))$ is put in the token field, $tk(i, m).Ltime$, before the token begins its second circuit. The timestamp pair, $tk(i, m).ts$ and $tk(i, m).Ltime$, is stored at each process and used to determine whether to accept incoming messages. A message whose timestamp exceeds $tk(i, m).ts$ and is less than $tk(i, m).Ltime$ is discarded. If the sender is p_i , then the timestamp in the message must be greater than $tk(i, m).Ltime$, or the message is discarded. All other messages are accepted.

Each process must wait until it has received the token a second time before it sends any messages. Any communication event of the underlying computation with timestamp greater than $T(LastEvent(f_i^m))$ and less than $T(c1_i(i, m))$ must be considered a potential orphan. There is no way to determine, based on timestamp, whether a message is sent after a process has been rolled back and is valid, or whether it is an orphan, if its timestamp falls in this interval. To prevent valid messages from being incorrectly discarded or eliminated through rollback, the restriction on communication is necessary. Each process must also buffer any incoming message that arrives after $c1_j(i, m)$ and whose timestamp is greater than $tk(i, m).ts$ until $c2_j(i, m)$ has occurred. The reason for this is that a rolled back process cannot determine if such a message originated in an orphan event until the value of $tk(i, m).Ltime$ is received.

4.5.2 Formal Specification

Synchronous Recovery Protocol - Two Wave

TRB.1 The occurrence of rs_i^m implies

$$\begin{aligned}
& LastEvent(f_i^m) \Rightarrow rs_i^m \wedge \\
& tk(i, m).ts = T(LastEvent(f_i^m)) \wedge \\
& tk(i, m).id = i \wedge \\
& tk(i, m).times(j) = T(s) \text{ where } \exists \eta(s') \text{ such that } \eta(s) \Rightarrow \eta(s') \Rightarrow rs_i^m \wedge \\
& \sigma(s) = \sigma(s')
\end{aligned}$$

A restart event occurs when the latest event that occurred prior to failure is recovered from stable storage. A token incorporating the timestamp of the restart event, the id of the recovering process, and a vector of latest message receipt times is created during this event.

TRB.2. $wl_i(i, m)$ occurs iff

$$\begin{aligned}
& \exists rs_i^m \text{ such that } rs_i^m \Rightarrow wl_i(i, m) \wedge \\
& \exists ck_i' \text{ such that } ck_i' \Rightarrow wl_i(i, m) \wedge CK(ck_i', rs_i^m) \wedge \\
& \left[\begin{array}{l} \exists e_i' \text{ such that } rs_i^m \Rightarrow e_i' \Rightarrow wl_i(i, m) \wedge \\ e_i' \text{ is an event of the underlying computation.} \end{array} \right]
\end{aligned}$$

A formerly failed process creates and propagates a token, event $wl_i(i, m)$, immediately after the occurrence of a restart event rs_i^m .

TRB.3. A rollback event, rb_j' , is instigated by rs_i^m iff

$$\begin{aligned}
& cl_j(i, m) \Rightarrow rb_j' \wedge \\
& \left[\begin{array}{l} \exists e_j' \text{ such that } cl_j(i, m) \Rightarrow e_j' \Rightarrow rb_j' \wedge \\ e_j' \text{ is an event of the underlying computation.} \end{array} \right]
\end{aligned}$$

Rollback is immediately instigated by the arrival of the token.

TRB.4. The occurrence of rb_j' instigated by rs_i^m implies

$$\begin{aligned}
& e_j' \Rightarrow cl_j(i, m) \text{ iff } T(e_j') < tk(i, m).ts \wedge \\
& Retransmit(tk(i, m).ts, tk(i, m).id, tk(i, m).times, j, cl_j(i, m)).
\end{aligned}$$

Rollback implies that any event that occurs before the $cl_j(i, m)$ and whose timestamp is greater than the token, is eliminated.

TRB.5. $w1_j(i, m), i \neq j$ occurs iff

$$\begin{aligned} & \exists rb'_j \text{ instigated by } rs_i^m \text{ such that } rb'_j \Rightarrow w1_j(i, m) \wedge \\ & \exists ck'_j \text{ such that } ck'_j \Rightarrow w1_j(i, m) \wedge CK(ck'_j, rb'_j) \wedge \\ & \left[\begin{array}{l} \nexists e'_j \text{ such that } rb'_j \Rightarrow e'_j \Rightarrow w1_j(i, m) \wedge \\ e'_j \text{ is an event of the underlying computation.} \end{array} \right] \end{aligned}$$

A non-failed process will propagate the token only after it has rolled back and checkpointed its state.

TRB.6. $cl_i(i, m) \Rightarrow w2_i(i, m)$.

The second polling wave begins when the first wave is completed.

TRB.7. The occurrence of $w2_i(i, m)$ implies that $tk(i, m).Ltime = T(cl_i(i, m))$.

The time of the last event of the first polling wave is put in the token before the beginning of the second polling wave.

TRB.8. The occurrence of $\eta(s)$ where $\rho(s) = p_j$, and $w1_j(i, m) \Rightarrow \eta(s)$ implies that $w2_j(i, m) \Rightarrow \eta(s)$.

A process will not accept any incoming messages until the second polling wave arrives.

TRB.9 The occurrence of $\eta(s)$ where $w2_{\rho(s)}(i, m) \Rightarrow \eta(s)$ implies that

$$\begin{aligned} & T(s) < tk(i, m).ts \vee \\ & T(s) > tk(i, m).Ltime. \end{aligned}$$

Any message arriving after a polling wave must be compared to the token timestamp and the timestamp of $cl_i(i, m)$ to determine whether it originated in an orphan event.

TRB.10 The occurrence of s such that $w1_{\sigma(s)}(i, m) \Rightarrow s$ implies that $c2_{\sigma(s)}(i, m) \Rightarrow s$.

Messages are not sent until the arrival of the second polling wave.

TRB.11 $w2_j(i, m), i \neq j$, occurs iff

$$\begin{aligned} & c2_j(i, m) \Rightarrow w2_j(i, m) \wedge \\ & \text{Logged}(c2_j(i, m)) \wedge \\ & \left[\begin{array}{l} \nexists e'_j \text{ such that } c2_j(i, m) \Rightarrow e'_j \Rightarrow w2_j(i, m) \wedge \\ e'_j \text{ is an event of the underlying computation.} \end{array} \right] \end{aligned}$$

TRB.12 The occurrence of $c2_i(i, m)$ implies $\text{Logged}(c2_i(i, m))$.

4.5.3 An Example - Synchronous Recovery

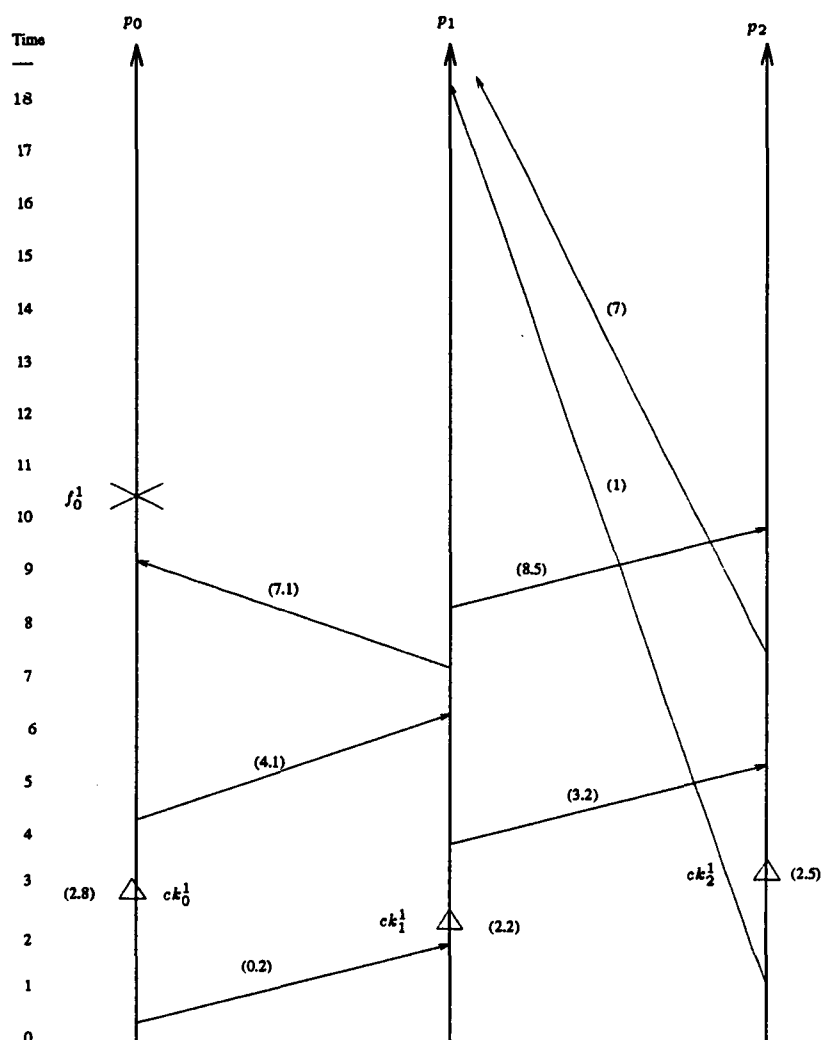


Figure 4.13: Synchronous Protocol: Example

Figure 4.13 shows a system of three processes. The processes take checkpoints at ck_0^1 , ck_1^1 , and ck_2^1 . Each event on a process time line is tagged with the clock time of its occurrence. Each message is tagged with $T(s)$, where $T(s)$ is the clock time of the send. Process p_0 fails just after the receiving a message at (9.0).

Figure 4.14 shows the events that occur during rollback. Upon restart of p_0 , the check-

point ck_0^1 is restored, and the restart event rs_0^1 is performed by the protocol. A token, with values:

- $tk(0,1).ts = 2.8$
- $tk(0,1).id = 0$
- $tk(0,1).times = \langle 0, 0, 0 \rangle$
- $tk(0,1).Ltime = Null$

is created and propagated to p_1 (the dashed lines indicate token transmission). Upon receipt of the token, p_1 rolls back to the latest checkpoint whose time is not greater than $tk(i,m).ts$. Hence p_1 rolls back to its state at time (2.2). The polling event $c1_1(0,1)$ is reinstated, the rollback event rb_1^1 occurs, the state of p_1 at rb_1^1 is saved in stable storage, and the token is sent to p_2 . Process p_2 takes action similar to p_1 to roll back to time (2.5). The token is then returned to p_0 . When the token returns to p_0 , $tk(i,m).Ltime$ is set to (15), the timestamp of the arrival of the token at p_0 . The token is then circulated again. Completion of the protocol occurs at event $c2_0(0,1)$.

Two messages are in transit while the polling wave is taking place. The message from p_2 to p_1 with label (1) will be accepted when it arrives because (1) $\not\prec$ (2.8). Application of Rule TRB.9 will result in message (7.0) being discarded when it arrives at p_1 .

The message at (3.2) from p_1 to p_2 will be eliminated by rollback even though it is not an orphan. This is an example of how a real time based protocol may rollback the system further than necessary. The protocol is correct: the resulting state is consistent, and every message that originates in a non-lost event is eventually received, but the rolled back state is not optimal.

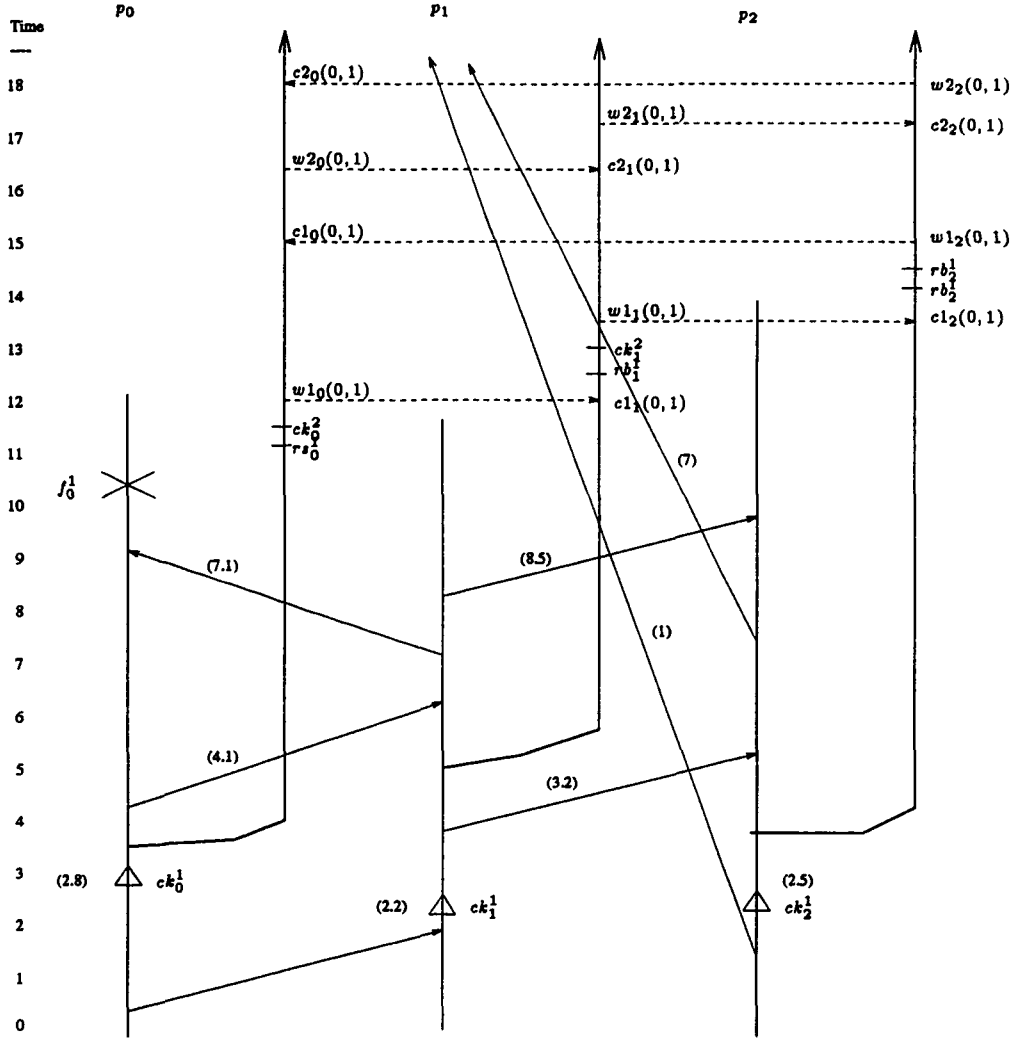


Figure 4.14: Resulting Consistent State

4.5.4 Correctness

This protocol is designed to work in a system environment in which failures are serial. The failure restrictions presented in Section 4.4.4 are modified below to apply to a two wave protocol:

$$\begin{aligned}
& e'_i \Rightarrow f_i^m, \text{ for all } e'_i \in FW(j, k) \wedge \\
& \neg Orphan(e'_x, f_i^m), \text{ for all } e'_x \in FW(j, k) \wedge \\
& w2_l(j, k) \Rightarrow c1_l(i, m), \text{ for all } p_l \in \Pi \wedge \\
& c2_j(j, k) \Rightarrow c1_j(i, m),
\end{aligned}$$

or

$$\begin{aligned}
& e'_j \Rightarrow f_j^k, \text{ for all } e'_j \in FW(i, m) \wedge \\
& \neg Orphan(e'_x, f_j^k), \text{ for all } e'_x \in FW(i, m) \wedge \\
& w2_l(i, m) \Rightarrow c1_l(j, k), \text{ for all } p_l \in \Pi \wedge \\
& c2_i(i, m) \Rightarrow c1_i(j, k).
\end{aligned}$$

The following result establishes that the protocol allows no causal links to be created between rs_i^m and any event that does not happen after $c1_i(i, m)$.

Lemma 45 *For any event e'_j , if $e'_j \Rightarrow c1_j(i, m)$ then $rs_i^m \not\Rightarrow e'_j$.*

Proof: Assume the contrary. Let e'_j be an event on p_j such that $rs_i^m \Rightarrow e'_j \Rightarrow c1_j(i, m)$. The token cannot establish this causal link directly, because e'_j precedes the arrival of the token. Therefore, there must exist a send s such that $w1_{\sigma(s)}(i, m) \Rightarrow s \Rightarrow e'_j$.

Initially consider the case where $e'_j = \eta(s)$. This implies that $w1_{\sigma(s)}(i, m) \Rightarrow s \Rightarrow \eta(s) \Rightarrow c1_j(i, m)$. Because $c1_j(i, m) \Rightarrow c2_k(i, m)$ for all k , it is also the case that $w1_{\sigma(s)}(i, m) \Rightarrow s \Rightarrow c2_{\sigma(s)}(i, m)$. This contradicts Rule TRB.10 of the protocol.

If $e'_j \neq \eta(s)$ then there exists a series of message events between s and e'_j . At some point in this sequence of events a message there must exist s' and $\eta(s')$, such that $\eta(s') \Rightarrow e'_j \Rightarrow c1_j(i, m)$, and $w1_{\sigma(s)}(i, m) \Rightarrow s' \Rightarrow \eta(s')$. This implies $w1_{\sigma(s)}(i, m) \Rightarrow s' \Rightarrow c1_i(i, m)$, contradicting TRB.10. ■

The following lemma shows that the timestamp of any messages lost due to a failure f_i^m must be greater than the timestamp of the restart event.

Lemma 46 $\forall e'_i$ such that $Disc(e_i, rs_i^m)$, $T(LastEvent(f_i^m)) < T(e'_i)$.

Proof: If $Disc(e'_i, rs_i^m)$ then $LastEvent(f_i^m) \Rightarrow e'_i$ in the failed execution history. Therefore, $T(LastEvent(f_i^m)) < T(e'_i)$. ■

Lemma 47 shows that every orphaned event has a timestamp greater than the token timestamp.

Lemma 47 If $Orphan(e'_j, f_i^m)$ then $tk(i, m).ts < T(e'_j)$.

Proof: By the hypothesis, $Orphan(e'_j, f_i^m)$. Then there exists e'_i such that $Disc(e'_i, rs_i^m)$, and $e'_i \Rightarrow e'_j$. $Disc(e'_i, rs_i^m)$ implies $T(LastEvent(f_i^m)) < T(e'_i)$, Lemma 46. Therefore, $T(LastEvent(f_i^m)) < T(e'_i) < T(e'_j)$. Because $tk(i, m).ts = T(LastEvent(f_i^m))$, $tk(i, m).ts < T(e'_j)$. ■

We use Lemma 47, to show that the token transmission event of the first wave is never an orphan.

Lemma 48 For any $w1_j(i, m)$ event as it is specified in the Synchronous Recovery Protocol - Two Wave, $\neg Orphan(w1_j(i, m), f_i^m)$.

Proof: Assume the contrary, an event $w1_j(i, m)$ exists for which $Orphan(w1_j(i, m), f_i^m)$. Then there exists e'_i such that $Disc(e'_i, rs_i^m)$, and $e'_i \Rightarrow w1_j(i, m)$. Also let $w1_j(i, m)$ be the earliest token transmission event in the polling wave for which $Orphan(w1_j(i, m), f_i^m)$. By Rules TRB.3 and TRB.5, $c1_j(i, m) \Rightarrow rb_i^k \Rightarrow w1_j(i, m)$. So, $Orphan(w1_j(i, m), f_i^m)$ implies $Orphan(c1_j(i, m), f_i^m)$. Since $w1_j(i, m)$ is the earliest token departure event where

$Orphan(w1_j(i, m), f_i^m), c1_j(i, m)$ must be the earliest token arrival event for which $Orphan(c1_j(i, m), f_i^m)$. Therefore, $\neg Orphan(c1_{j-1(mod N)}(i, m), f_i^m)$. Then there must exist e'_j such that $e'_i \Rightarrow e'_j \Rightarrow c1_j(i, m) \Rightarrow w1_j(i, m)$. This implies that $Orphan(e'_j, f_i^m)$ is true, and by Lemma 47, $tk(i, m).ts < T(e'_j)$. This contradicts Rules TRB.4 and TRB.5 of the rollback protocol. ■

Lemma 49 *If $Disc(e'_i, rs_i^m)$ then $T(e'_i) < tk(i, m).Ltime$.*

Proof: Assume the contrary. If $T(e'_i) \geq tk(i, m).Ltime$ then either $e'_i = c1_i(i, m)$, or $c1_i(i, m) \Rightarrow e'_i$. In either case, $rs_i^m \Rightarrow e'_i$ contradicting our initial hypothesis that $Disc(e'_i, rs_i^m)$. ■

Lemma 50 *If $\eta(s) \Rightarrow w1_{\rho(s)}(i, m) \vee w1_{\rho(s)}(i, m) \Rightarrow \eta(s)$ then $\neg Orphan(s, f_i^m) \wedge \neg Disc(s, rs_i^m)$.*

Proof: Case 1: Assume $\eta(s) \Rightarrow w1_{\rho(s)}(i, m)$. By Lemma 48 $\neg Orphan(w1_{\rho(s)}(i, m), f_i^m)$. Therefore, $\neg Orphan(\eta(s), f_i^m)$, $\neg Orphan(s, f_i^m)$, and $\neg Disc(s, rs_i^m)$. By Rules TRB.8 and TRB.10, $s \Rightarrow rs_i^m$, or $s \Rightarrow c1_{\sigma(s)}(i, m)$. We have shown that $\neg Orphan(s, f_i^m)$, so in either case, $tk(i, m).ts > T(s)$. Therefore, $s \Rightarrow w1_{\sigma(s)}(i, m)$, and $\neg Disc(s, w1_{\sigma(s)}(i, m))$. Case 2: Assume that $w1_{\rho(s)}(i, m) \Rightarrow \eta(s)$. Also assume that there does not exist $\eta(s')$ such that $\eta(s') \Rightarrow \eta(s)$, and $w1_{\rho(s')}(i, m) \Rightarrow \eta(s')$. In other words assume that $\eta(s)$ is the earliest event, in this causal chain, that occurs following the wave. The following proves that if $Orphan(s, f_i^m)$ or $Disc(s, rs_i^m)$, then rule TRB.9 is violated.

First, consider $Disc(s, w1_i(i, m))$. This implies $Disc(s, rs_i^m)$, Rule TRB.2. Lemma 46 shows that $tk(i, m).ts < T(s)$. This violates the first disjunct of TRB.9. Lemma 49 shows that it is not possible for $T(s) > tk(i, m).Ltime$ thus violating the last disjunct of TRB.9.

In the case that $\sigma(s) \neq i$, $Disc(s, w1_{\sigma(s)}(i, m))$ implies $tk(i, m).ts > T(s)$ (Rule CRB.4), and $Orphan(s, f_i^m)$. In addition, $Orphan(s, f_i^m)$ implies $Disc(s, w1_{\sigma(s)}(i, m))$, Lemma 48. Therefore, it will suffice to show that $Orphan(s, f_i^m)$ leads to a contradiction.

Now assume $Orphan(s, f_i^m)$. This implies that $T(s) > tk(i, m).ts$ (by Lemma 47), thus violating the first disjunct of TRB.9. $T(s) > tk(i, m).Ltime$ implies $T(w1_{\sigma(s)}(i, m)) < T(s)$. This, in turn, implies $w1_{\sigma(s)}(i, m) \Rightarrow s$, contradicting Lemma 48 which states that $\neg Orphan(w1_j(i, m), f_i^m)$ for all $p_j \in \Pi$. Thus neither of the two disjuncts of TRB.9 can be satisfied. ■

Lemma 51 *If $\neg Disc(s, w_{\sigma(s)}(i, m))$ then $\eta(s) \Rightarrow w1_{\rho(s)}(i, m) \vee w1_{\rho(s)}(i, m) \Rightarrow \eta(s)$.*

Proof:

Case 1: $s \Rightarrow w1_{\sigma(s)}(i, m)$. This implies $T(s) < tk(i, m).ts$. If the receipt of s is lost by a failed process it will be retransmitted during the rollback process (Rule TRB.4). If $\eta(s)$ arrives after the token it will be accepted, because the first disjunct of TRB.9 will be satisfied. If $\eta(s)$ occurs before the arrival of the token, $\eta(s) \Rightarrow w1_{\rho(s)}(i, m)$ (Rule TRB.4).

Case 2: $w1_{\sigma(s)}(i, m) \Rightarrow s$. By Rule TRB.8, this implies $w2_{\sigma(s)}(i, m) \Rightarrow s$. Hence, $T(s) > tk(i, m).Ltime$. Therefore, $\eta(s)$ will be accepted, and $w1_{\rho(s)}(i, m) \Rightarrow \eta(s)$. ■

Theorem 20 *The completion of a valid wave in the Synchronous Rollback protocol satisfies RB.*

Proof: Lemma 48 showed that $\neg Orphan(w1_j(i, m), f_i^m)$ for all $p_j \in \Pi$. Lemma 50 showed that there does not exist $\eta(s)$ such that $w1_j(i, m) \Rightarrow \eta(s)$, and $Orphan(\eta(s), f_i^m)$. Therefore, $\neg Orphan(w2_j(i, m), f_i^m)$ for all $w2_j(i, m) \in FW(i, m)$ thus satisfying RB(a). Since $w1_j(i, m) \Rightarrow w2_j(i, m)$ for all $p_j \in \Pi$ and Lemmas 50 and 51 have shown that RB(b) and RB(c) hold for all $w1_j(i, m)$, clearly, RB(b) and RB(c) hold for all $w2_j(i, m)$. ■

4.6 Causal Recovery Protocol - Two Wave

The causal protocol presented in this section is modelled after the two wave synchronous protocol presented in Section 4.5.1. In it the recovering process generates a token that

traverses the virtual ring of processes twice to create two polling waves. During the first wave the token is used to identify and rollback orphan events. The token is used in the second wave to transmit the timestamp of the last event in the first wave to every process. This information is used to identify and discard any orphan messages that are in transit during recovery. The polling events of this protocol are the same as those defined for the Synchronous - Two Wave protocol.

Vector timestamps are associated with each application event instead of real time values. Unlike the causal protocol presented in Section 4.4 polling events cause vector time to increase just as events of the underlying computation.

In this protocol, as in the Synchronous - Two Wave protocol, and unlike the Causal Recovery Protocol - Single Wave, process rollback does not cause vector time to go backward. Every event in the polling waves are causally related and their vector timestamps reflect this relationship. This eliminates the need for incarnation numbers. The timestamps of wave events can be used instead.

4.6.1 Informal Description

Recovery of a failed process follows the same pattern as the previously described protocols. The process retrieves its latest checkpoint from stable storage, replays messages from the stable log, and instigates a restart event. The vector clock of the recovering process is recovered as part of the checkpointed state and is updated as messages are replayed and the the restart event occurs.

At the time of a restart event, the only knowledge that a failed process has of any lost event is that its vector clock value must be at least as great as the vector clock value of the restart event. Before the recovering process communicates with any other process, i.e., before any causal links can be established between rs_i^m and any other event, it is true that $rs_i^m \not\Rightarrow e'_j$ for all e'_j . Therefore, if there is some event, e'_j in the system for which $V_j(e'_j) > V_i(rs_i^m)$, then there must be some lost event e'_i in p_i for which $e'_i \Rightarrow e'_j$. In other

words, any event in the system for which $V_j(e'_j) > V_i(rs_i^m)$ is an orphan event. We could not make a similar claim for the synchronous protocol; that $T(e'_j) > T(\text{LastEvent}(f_i^m))$ implied e'_j was an orphan event. The link between the ordering of real clock values and the partial order is just not strong enough.

Note how the loss of events disrupts the normal isomorphism between vector time and causality. For all orphan events in the system $V_i(rs_i^m) < V_j(e'_j)$, but $rs_i^m \not\Rightarrow e'_j$. We can use this breakdown in the isomorphism to identify orphan events if we are careful not to introduce causal links between rs_i^m and any event e'_j for which $\text{Orphan}(e'_j, f_i')$.

To do this, we require that no communication that causally follows rs_i^m , i.e., no send event s for which $rs_i^m \Rightarrow s$, is received before the first polling wave arrives at a process. The causal chain established by the first circulation of the token must act as a demarcation line, so that for any event before the wave, $e'_j \Rightarrow \text{cl}_j(i, m)$, $V_i(rs_i^m) < V_j(e'_j)$ implies $rs_i^m \not\Rightarrow e'_j$. For any event after $\text{PW1}(i, m)$ the isomorphism of vector time to causality must hold, i.e., $V_i(e'_i) < V_j(e'_j)$ iff $e'_i \Rightarrow e'_j$.

In the synchronous protocol, preservation of the first polling wave as a line of demarcation required no process communication during the polling waves, except for retransmission and propagation of the token. In this protocol the restriction on process communication is not as stringent. Some process communication is allowed, so long as it doesn't establish a causal link between rs_i^m and an orphan event.

The failed process always instigates the polling wave. Once the restart event occurs, a token is generated containing the process id and the vector timestamp of the restart event. The token is composed of four fields:

- $tk(i, m).ts = V_i(rs_i^m)$
- $tk(i, m).id = i$
- $tk(i, m).Ltime = V_i(\text{cl}_i(i, m))$

- $tk(i, m).seq = V.seq_i$

The function of the token in the first wave is the same in this protocol as it is in the Causal Recovery Protocol - Single Wave and the Synchronous Recovery Protocol. When the token arrives at a process, p_j , the token's timestamp is compared to the current timestamp of p_j . If the timestamp in the token is less than p_j 's vector timestamp, then some of the events in p_j have been orphaned, and p_j must be rolled back. To rollback, the latest checkpoint, ck_j^k , such that $V_j(ck_j^k) \not\geq tk(i, m).ts$ is reinstated from stable storage, and any events with a timestamp not greater than the token timestamp are replayed from the logs.

During rollback p_j must retransmit any message that p_i has lost due to failure. P_j must also request retransmission of any messages it had received that originated in non-orphan events, but were eliminated through rollback. The rules for doing this are quite similar to those used in the single wave causal recovery protocol:

Retransmit($tk.ts, tk.id, tk.seq, id, c.event$)

For all $e'_{id} = \eta(s)$ such that:

$$\begin{aligned} \eta(s) &\Rightarrow c.event \wedge \\ V_{id}(\eta(s)) &> tk.ts \wedge \\ V_{\sigma(s)}(s) &\not\geq tk.ts \end{aligned}$$

retransmission of message from $p_{\sigma(s)}$ is requested

For all $e_{id} = s$ such that:

$$\begin{aligned}
s &\Rightarrow c.event \wedge \\
V_i^i(s) &> tk.seq(i) \wedge \\
V_i(s) &\not\prec tk.ts
\end{aligned}$$

s is retransmitted to $p_{tk.id}$

Once rollback is completed, the arrival of $c1_j(i, m)$ is replayed. The state of p_j is updated to include the polling event, and $V_j(p_j)$ is updated accordingly. This is done to preserve the causal order between polling events and to prevent “time” from going backwards.

As in the synchronous protocol, after $c1_j(i, m)$ is replayed, rb_j' occurs to indicate rollback is complete. A checkpoint of process p_j 's state is taken before the token is propagated. Once the process has been rolled back, it may resume normal operations. However, p_j may not send a message that could arrive ahead of the first polling wave. An event, e'_k , occurs ahead of wave $PW1(i, m)$ if $e'_k \Rightarrow w1_k(i, m)$. If e'_j is a send event, and $\eta(e'_j) \Rightarrow w1_k(i, m)$, then p_j has sent a message that arrived ahead of $PW1(i, m)$. If $c1_j(i, m) \Rightarrow e'_j$, then this message has “crossed” $PW1(i, m)$. To avoid sending such messages, p_j is restricted from sending any message to any process p_k where $c1_k(i, m) \not\prec c1_j(i, m)$. This restriction prevents the establishment of a causal link between rs_i^m and events ahead of the first polling wave. Any such link would disrupt the relationship between vector clock values and orphan events.

One purpose of the second polling wave is to make each process aware that $PW1(i, m)$ is complete. The arrival of $c2_j(i, m)$ implies $c1_k(i, m) \Rightarrow c1_j(i, m)$ for all $p_k \in \Pi$. The occurrence of $c2_j(i, m)$ means p_j can send a message to any process.

As in all the previous protocols the token is propagated from p_j to $p_{j+1(mod N)}$ until it returns to the originating process p_i . When this occurs every process has been returned to a consistent state. Messages originating in orphaned send events must be discarded upon

arrival to insure that the system state remains consistent. This is accomplished through the use of the vector clock value of $cl_i(i, m)$. A message sent during an orphan event must have been sent ahead of $PW1(i, m)$. Therefore, $V_{\sigma(s)}(s) < V_{\sigma(s)}(w1_{\sigma(s)}(i, m)) < V_i(cl_i(i, m))$. It is also true that $tk(i, m).ts < V_{\sigma(s)}(s)$ if $Orphan(f_i^m, s)$. Using $tk(i, m).ts$ and $V_i(cl_i(i, m))$, every process can determine whether to accept messages that arrives after the first polling wave.

The second polling wave disseminates $V_i(cl_i(i, m))$ to all the processes. $V_i(cl_i(i, m))$ is put in the token field, $tk(i, m).Ltime$, before the token begins its second circuit. The vector timestamp pair, $tk(i, m).ts$ and $tk(i, m).Ltime$, is stored at each process and used to determine whether to accept incoming messages. A message whose timestamp exceeds $tk(i, m).ts$ and is less than $tk(i, m).Ltime$ is discarded. If the sender is p_i , then the timestamp in the message must be greater than $tk(i, m).Ltime$, or the message is discarded. All other messages are accepted.

The only process which must freeze during this protocol is the failed process. It must buffer any message received until $cl_i(i, m)$ has occurred. All other processes must buffer any incoming message whose timestamp is greater than $tk(i, m).ts$ until $c2_j(i, m)$ has occurred. The reason for this is that a rolled back process cannot determine if such a message originated in an orphan event until the value of $tk(i, m).Ltime$ is received.

4.6.2 Formal Specification

Causal Recovery Protocol: Two Wave - Serial Failure

CRB2.1 The occurrence of rs_i^m implies

$$\begin{aligned} LastEvent(f_i^m) &\Rightarrow rs_i^m \wedge \\ tk(i, m).ts &= V_i(rs_i^m) \wedge \\ tk(i, m).id &= i. \end{aligned}$$

A restart event occurs when the latest event that occurred prior to failure is recovered from stable storage. A token incorporating the timestamp of the restart event and the id of the recovering process is created during this event.

CRB2.2. $w1_i(i, m)$ occurs iff

$$\begin{aligned} & \exists rs_i^m \text{ such that } rs_i^m \Rightarrow w1_i(i, m) \wedge \\ & \exists ck_i' \text{ such that } ck_i' \Rightarrow w1_i(i, m) \wedge CK(ck_i', rs_i^m) \wedge \\ & \left[\begin{array}{l} \exists e_i' \text{ such that } rs_i^m \Rightarrow e_i' \Rightarrow w1_i(i, m) \wedge \\ e_i' \text{ is an event of the underlying computation.} \end{array} \right] \end{aligned}$$

A formerly failed process creates and propagates a token, event $w1_i(i, m)$, immediately after the occurrence of a restart event rs_i^m .

CRB2.3. An rollback event, rb_j' , is instigated by rs_i^m iff

$$\begin{aligned} & c1_j(i, m) \Rightarrow rb_j' \wedge \\ & \left[\begin{array}{l} \exists e_j' \text{ such that } c1_j(i, m) \Rightarrow e_j' \Rightarrow rb_j' \wedge \\ e_j' \text{ is an event of the underlying computation.} \end{array} \right] \end{aligned}$$

Rollback is instigated by the arrival of the token.

CRB2.4. The occurrence of rb_j' instigated by rs_i^m implies

$$\begin{aligned} & e_j' \Rightarrow c1_j(i, m) \text{ iff } V_j(e_j') \not\prec tk(i, m).ts \wedge \\ & Retransmit(tk(i, m).ts, tk(i, m).id, tk(i, m).seq, j, c1_j(i, m)). \end{aligned}$$

Rollback implies that any event that occurs before the $c1_j(i, m)$ and whose timestamp is greater than the token, is eliminated.

CRB2.5. $w1_j(i, m), i \neq j$ occurs iff

$$\begin{aligned} & \exists rb_j' \text{ associated with } rs_i^m \text{ such that } rb_j' \Rightarrow w1_j(i, m) \wedge \\ & \exists ck_j' \text{ such that } ck_j' \Rightarrow w1_j(i, m) \wedge CK(ck_j', rb_j') \wedge \\ & \left[\begin{array}{l} \exists e_j' \text{ such that } rb_j' \Rightarrow e_j' \Rightarrow w1_j(i, m) \wedge \\ e_j' \text{ is an event of the underlying computation.} \end{array} \right] \end{aligned}$$

A non-failed process will propagate the token only after it has rolled back.

CRB2.6. $c1_i(i, m) \Rightarrow w2_i(i, m)$

The second polling wave begins when the first wave is completed.

CRB2.7. The occurrence of $w2_i(i, m)$ implies that $tk(i, m).Ltime = V_i(c1_i(i, m))$.

The vector time of the last event of the first polling wave is put in the token before the beginning of the second polling wave.

CRB2.8. The occurrence of $\eta(s)$ where $\rho(s) = p_i$, and $rs_i^m \Rightarrow \eta(s)$, implies that $w2_i(i, m) \Rightarrow \eta(s)$.

A recovering process will not accept any incoming messages until the first polling wave is completed.

CRB2.9 The occurrence of $\eta(s)$ where $w1_{\rho(s)}(i, m) \Rightarrow \eta(s)$ implies that

$$\begin{aligned} &V_{\sigma(s)}(s) \not\geq tk(i, m).ts \vee \\ &V_{\sigma(s)}(s) > tk(i, m).Ltime \vee \\ &V_{\sigma(s)}(s) \parallel tk(i, m).Ltime \wedge \sigma(s) \neq i. \end{aligned}$$

Any message arriving after a polling wave must be compared to the token timestamp and the timestamp of $cl_i(i, m)$ to determine whether it originated in an orphan event.

CRB2.10 The occurrence of s , such that $w1_{\sigma(s)}(i, m) \Rightarrow s \Rightarrow c2_{\sigma(s)}(i, m)$, implies that $cl_{\rho(s)}(i, m) \Rightarrow \eta(s)$.

Messages are not allowed to cross the first polling wave.

CRB2.11 $w2_j(i, m)$, $i \neq j$, occurs iff

$$\begin{aligned} &c2_j(i, m) \Rightarrow w2_j(i, m) \wedge \\ &\quad Logged(c2_j(i, m)) \wedge \\ &\left[\begin{array}{l} \exists e'_j \text{ such that } c2_j(i, m) \Rightarrow e'_j \Rightarrow w2_j(i, m) \wedge \\ e'_j \text{ is an event of the underlying computation.} \end{array} \right] \end{aligned}$$

CRB2.12 The occurrence of $c2_i(i, m)$ implies $Logged(c2_i(i, m))$.

4.6.3 An Example

In Figure 4.15 we see a system of three processes. The processes take checkpoints at ck_0^1 , ck_1^1 , and ck_2^1 . Each event on a process time line is tagged with the vector time of its occurrence. Each message is tagged with (x, y, z) , where (x, y, z) is the vector time of the send. Process p_0 fails just after the message receipt which increments its vector clock to $(4, 5, 0)$.

Figure 4.16 shows the system during execution of the protocol and when rollback is complete. Upon restart of p_0 , the checkpoint ck_0^1 is restored, and the restart event rs_0^1 is performed by the protocol. A token, with timestamp of $(3, 0, 0)$ is created and propagated to p_1 (the dashed lines indicate token transmission). Upon receipt of the token, p_1 rolls back to a point meeting the requirement that its vector time is not greater than $(3, 0, 0)$. Hence

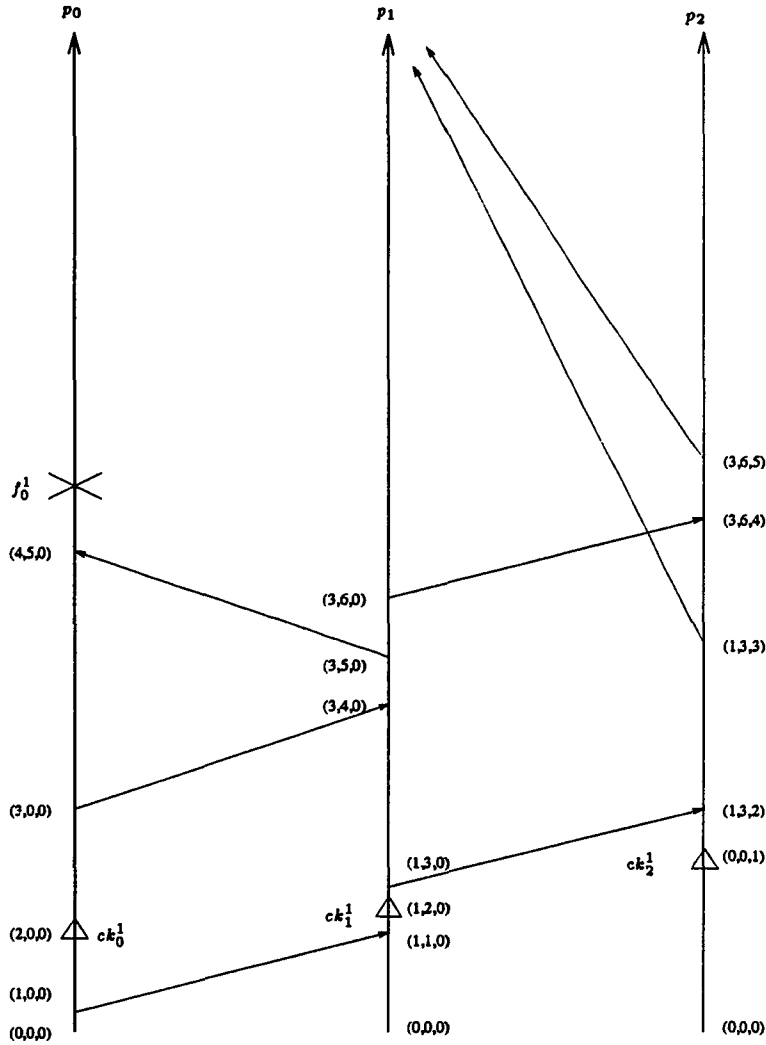


Figure 4.15: An Example

p_1 rolls back to its state at time $(1, 3, 0)$. Process p_1 retransmits the message timestamped $(3, 5, 0)$ because the second element of the timestamp of the message is greater than the second element of $tk(0, 1).seq$, meaning that p_0 lost that message. The polling event $cl_1(0, 1)$ is reinstated, the rollback event rb_1^1 occurs, the state of p_1 at rb_1^1 is saved in stable storage, and the token is sent to p_2 . Process p_2 takes action similar to p_1 to roll back to time $(1, 3, 2)$. The token is then returned to p_0 . When the token returns to p_0 , $tk(i, m).Ltime$ is set to

(6, 10, 9), the timestamp of the arrival of the token at p_0 . The token is then circulated again. Completion of the protocol occurs at event $c2_0(0, 1)$.

Two messages are in transit while the polling wave is taking place. The message from p_2 to p_1 with label (1, 3, 3) will be accepted when it arrives because $(1, 3, 3) \not\prec (3, 0, 0)$. Application of Rule CRB2.10 will result in message (3, 6, 5) being discarded when it arrives at p_1 .

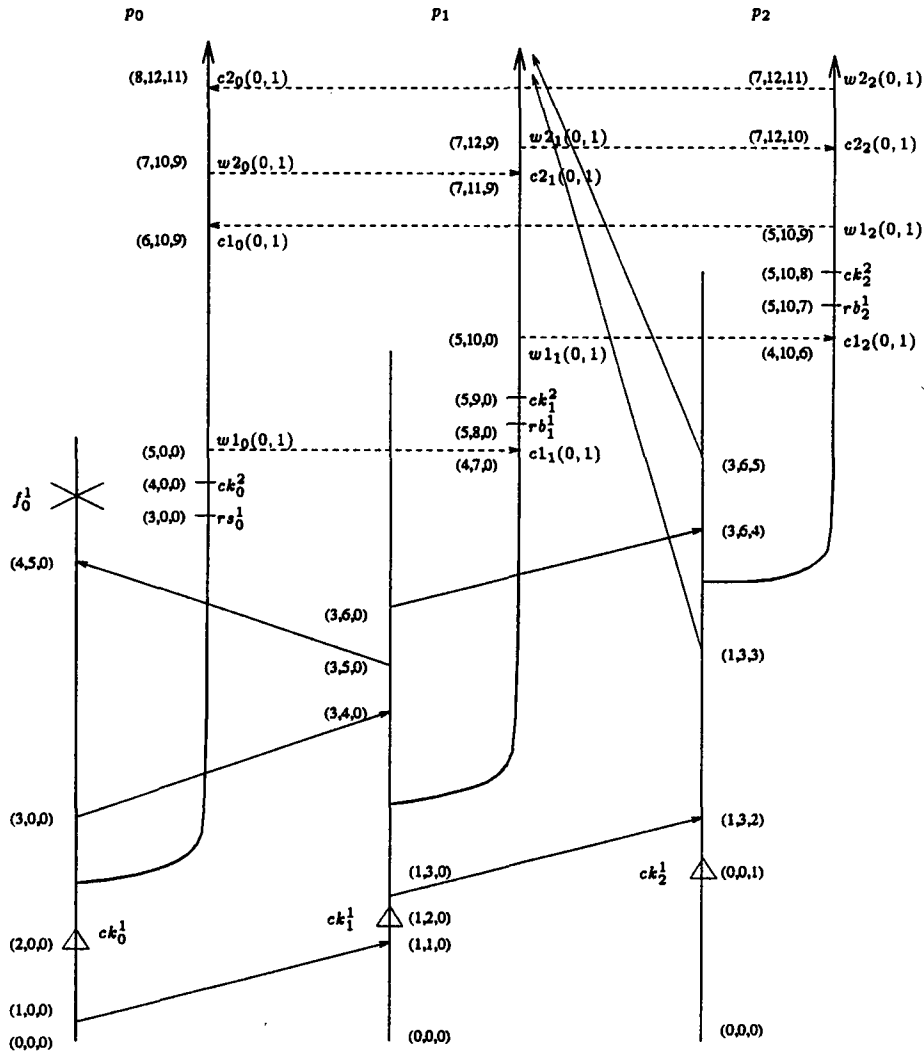


Figure 4.16: Resulting Consistent State

4.6.4 Correctness

The first step in showing the correctness of this protocol is to prove that no messages cross from behind the wave to the front of the wave.

Lemma 52 *For any event e'_j , if $e'_j \Rightarrow c1_j(i, m)$ then $rs_i^m \not\Rightarrow e'_j$.*

Proof: Assume the contrary. Let e'_j be an event on p_j such that $rs_i^m \Rightarrow e'_j \Rightarrow c1_j(i, m)$. The token cannot establish this causal link directly because e'_j precedes the arrival of the token. Therefore, there must exist a send s such that $w1_{\sigma(s)}(i, m) \Rightarrow s \Rightarrow e'_j$.

Initially consider the case where $e'_j = \eta(s)$. This implies that $w1_{\sigma(s)}(i, m) \Rightarrow s \Rightarrow \eta(s) \Rightarrow c1_j(i, m)$. Because $c1_j(i, m) \Rightarrow c2_k(i, m)$ for all k , it is also the case that $w1_{\sigma(s)}(i, m) \Rightarrow s \Rightarrow c2_{\sigma(s)}(i, m)$. This contradicts Rule CRB2.10 of the protocol.

If $e'_j \neq \eta(s)$ then there exists a series of message events between s and e'_j . At some point in this sequence of events a message must be sent to a process ahead of the wave from a process that has already propagated the token. So there exists s' and $\eta(s')$ such that $s \Rightarrow s' \Rightarrow \eta(s') \Rightarrow e'_j$, and $w1_{\sigma(s')}(i, m) \Rightarrow s' \Rightarrow \eta(s') \Rightarrow c1_{\rho(s')}(i, m)$. The above argument applies in this case as well. ■

Lemma 53 shows that the token timestamp as specified by the protocol is less than the timestamp of any orphaned event. Lemma 54 proves the converse for events occurring before the wave.

Lemma 53 *If $Orphan(e'_j, f_i^m)$ then $tk(i, m).ts < V_j(e'_j)$.*

Proof: By the hypothesis, $Orphan(e'_j, f_i^m)$. Then there exists e'_i such that $Disc(e'_i, rs_i^m)$ and $e'_i \Rightarrow e'_j$. By Lemma 36, $Disc(e'_i, rs_i^m)$ implies $V_i(rs_i^m) \leq V_i(e'_i)$. Therefore, $V_i(rs_i^m) \leq V_i(e'_i) < V_j(e'_j)$. Because $tk(i, m).ts = V_i(rs_i^m)$ (CRB2.1), $tk(i, m).ts < V_j(e'_j)$. ■

Lemma 54 *$\forall e'_j$ such that $e'_j \Rightarrow c1_j(i, m)$, if $tk(i, m).ts < V_j(e'_j)$ then $Orphan(e'_j, f_i^m)$.*

Proof: Suppose that $tk(i, m).ts < V_j(e'_j)$ for some event $e'_j \Rightarrow c1_j(i, m)$. This implies that $V_i(rs_i^m) < V_j(e'_j)$ (by CRB2.1). This in turn implies that there must exist at least one event e'_i such that $e'_i \Rightarrow e'_j$. Let e_i^k be the latest of the events in p_i such that $e_i^k \Rightarrow e'_i$. If this is the case then $V_i^i(e_i^k) = V_j^j(e'_j)$. The facts that $V_i(rs_i^m) < V_j(e'_j)$, and $V_i^i(e_i^k) = V_j^j(e'_j)$, imply that $V_i(rs_i^m) \leq V_i(e_i^k)$. Therefore, $e_i^k \not\Rightarrow rs_i^m$. However, it is also not possible that $rs_i^m = e_i^k$, or that $rs_i^m \Rightarrow e_i^k$, because in Lemma 52 we have shown that $e'_j \Rightarrow c1_j(i, m)$ implies $rs_i^m \not\Rightarrow e'_j$. Therefore, $Disc(e_i^k, rs_i^m)$. By definition, if $Disc(e_i^k, rs_i^m)$, and $e_i^k \Rightarrow e'_j$, then $Orphan(e'_j, f_i^m)$. ■

This result establishes the fact that the token, as constructed during the restoration of a formerly failed process, contains the information necessary to determine if any event is orphaned by failure.

Lemma 55 *For any $w1_j(i, m)$ event as specified in the Causal Recovery Protocol : Two Waves - Serial Failure, $\neg Orphan(w1_j(i, m), f_i^m)$.*

Proof: Assume the contrary, an event $w1_j(i, m)$ exists for which $Orphan(w1_j(i, m), f_i^m)$. Then there exists e'_i such that $Disc(e'_i, rs_i^m)$, and $e'_i \Rightarrow w1_j(i, m)$. Also let $w1_j(i, m)$ be the earliest token departure event in the polling wave for which $Orphan(w1_j(i, m), f_i^m)$. By Rule CRB2.3 and CRB2.5, $c1_j(i, m) \Rightarrow rb_i^k \Rightarrow w1_j(i, m)$. This implies $Orphan(c1_j(i, m), f_i^m)$. Since $w1_j(i, m)$ is the earliest token departure event where $Orphan(w1_j(i, m), f_i^m)$, $c1_j(i, m)$ must be the earliest token arrival event for which $Orphan(c1_j(i, m), f_i^m)$. Therefore, $\neg Orphan(c1_{j-1(mod N)}(i, m), f_i^m)$. Then there must exist e'_j such that $e'_i \Rightarrow e'_j \Rightarrow c1_j(i, m) \Rightarrow w1_j(i, m)$. This implies $Orphan(e'_j, f_i^m)$, and by Lemma 53, $tk(i, m).ts < V_j(e'_j)$. This contradicts Rules CRB2.4 and CRB2.5 of the rollback protocol. ■

Before we can show that $\neg Orphan(w2_j(i, m), f_i^m)$ for all $w2_j(i, m) \in FW(i, m)$ we must establish that orphaned messages in transit during or after a recovery are discarded.

Lemma 56 *If $\text{Disc}(e'_i, w1_i(i, m))$ then $V_i(e'_i) \not\geq tk(i, m).Ltime$.*

Proof: Assume the contrary. If $V_i(e'_i) > tk(i, m).Ltime$ then $V_i(e'_i) > V_i(c1_i(i, m))$ (Rule CRB2.7). Since $c1_j(i, m) \Rightarrow c1_i(i, m)$ for all $p_j \neq p_i \in \Pi$, this implies that $V_i(e'_i) > V_j(c1_j(i, m))$ for all $p_j \in \Pi$. However, if $\text{Disc}(e'_i, w1_i(i, m))$ then $c1_j(i, m) \not\geq e'_i$ (Rule CRB2.2), and $V_i^j(e'_i) < V_j^j(c1_j(i, m))$. If the j^{th} element of $c1_j(i, m)$'s vector clock value exceeds the j^{th} value of the timestamp of e'_i , then it is not possible for $V_i(e'_i) > V_j(c1_j(i, m))$. ■

Lemma 57 *If $\eta(s) \Rightarrow w1_{\rho(s)}(i, m) \vee w1_{\sigma(s)}(i, m) \Rightarrow \eta(s)$ then $\neg \text{Orphan}(s, f_i^m) \wedge \neg \text{Disc}(s, w1_{\sigma(s)}(i, m))$.*

Proof: Case 1: Assume $\eta(s) \Rightarrow w1_{\rho(s)}(i, m)$. By Lemma 55 $\neg \text{Orphan}(w1_{\rho(s)}(i, m), f_i^m)$. Therefore, $\neg \text{Orphan}(\eta(s), f_i^m)$, $\neg \text{Orphan}(s, f_i^m)$, and $\neg \text{Disc}(s, rs_i^m)$. By Rules CRB2.8 and CRB2.10, $s \Rightarrow rs_i^m$, or $s \Rightarrow c1_{\sigma(s)}(i, m)$. We have shown that $\neg \text{Orphan}(s, f_i^m)$, so in either case, $tk(i, m).ts \not\leq V_{\sigma(s)}(s)$. Therefore, $s \Rightarrow w1_{\sigma(s)}(i, m)$, and $\neg \text{Disc}(s, w1_{\sigma(s)}(i, m))$.

Case 2: Assume that $w1_{\rho(s)}(i, m) \Rightarrow \eta(s)$. Also assume that there does not exist $\eta(s')$ such that $\eta(s') \Rightarrow \eta(s)$ and $w1_{\rho(s')}(i, m) \Rightarrow \eta(s')$. In other words assume that $\eta(s)$ is the earliest event, in this causal chain, that occurs following the wave. The following proves that if $\text{Orphan}(s, f_i^m)$ or $\text{Disc}(s, w1_{\sigma(s)}(i, m))$, then rule CRB2.9 is violated.

First, consider $\text{Disc}(s, w1_i(i, m))$. Lemma 36 shows that $tk(i, m).ts \leq V_i(s)$. This violates the first disjunct of CRB2.9. Lemma 56 shows that it is not possible for $V_i(s) > tk(i, m).Ltime$. The last disjunct of CRB2.9 is violated because $\sigma(s) = i$.

In the case that $\sigma(s) \neq i$, $\text{Disc}(s, w1_{\sigma(s)}(i, m))$ implies $tk(i, m).ts < V_{\sigma(s)}(s)$ (Rule CRB2.4), and $\text{Orphan}(s, f_i^m)$. In addition, $\text{Orphan}(s, f_i^m)$ implies $\text{Disc}(s, w1_{\sigma(s)}(i, m))$, Lemma 55.

Therefore, assume $\text{Orphan}(s, f_i^m)$. This implies that $V_{\sigma(s)}(s) > tk(i, m).ts$ (Lemma 53), thus violating the first disjunct of CRB2.9. If $V_{\sigma(s)} > tk(i, m).Ltime$ then $c1_i(i, m) \Rightarrow s$.

This implies $w1_{\sigma(s)}(i, m) \Rightarrow s$, thus contradicting Lemma 55 in which it was shown that $\neg Orphan(w1_j(i, m), f_i^m)$ for all $p_j \in \Pi$. If $V_{\sigma(s)}(s) \parallel tk(i, m).Ltime$, and $\sigma(s) \neq i$, the final disjunct of CRB2.9 would be satisfied. However, this also implies $w1_{\sigma(s)}(i, m) \Rightarrow s$, contradicting Lemma 55. Thus neither of the last two disjuncts of CRB2.9 can be satisfied. ■

Lemma 58 *If $\neg Disc(s, w_{\sigma(s)}(i, m)) \wedge \neg Orphan(s, f_i^m)$ then $\eta(s) \Rightarrow w1_{\rho(s)}(i, m) \vee w1_{\rho(s)}(i, m) \Rightarrow \eta(s)$.*

Proof: $\neg Disc(s, w1_{\sigma(s)}(i, m))$ implies $s \Rightarrow w1_{\sigma(s)}(i, m)$, or $w1_{\sigma(s)}(i, m) \Rightarrow s$. $s \Rightarrow w1_{\sigma(s)}(i, m)$ implies $tk(i, m).ts \not\prec V_{\sigma(s)}(s)$, and $\neg Orphan(s, f_i^m)$. $w1_{\sigma(s)}(i, m) \Rightarrow s$ also implies $\neg Orphan(s, f_i^m)$, according to Lemma 55. Let s be a send such that $\neg Orphan(s, f_i^m)$.

Case 1: $s \Rightarrow w1_{\sigma(s)}(i, m)$. Lemmas 53 and 54 show that in this case $V_{\sigma(s)}(s) \not\prec tk(i, m).ts$. Therefore, the send is never eliminated during rollback. Given reliable channels the message will eventually arrive. The receipt of the message can only disappear from the causal order if it is lost by a failed process, rolled back by the protocol, or discarded upon arrival. The first possibility is that p_i (the failed process) lost the message due to its failure. Note that in this case $\rho(s) = i$. During the rollback at $p_{\sigma(s)}$ this message will be retransmitted. The occurrence of the rb event associated with $w1_{\sigma(s)}(i, m)$ guarantees this because $V_{\sigma(s)}^{\sigma(s)} > tk(i, m).seq(\sigma(s))$ (Rule CRB2.4). Therefore, $w1_i(i, m) \Rightarrow \eta(s)$. The second possibility is that $\eta(s) \Rightarrow c1_{\rho(s)}(i, m)$ and $\eta(s)$ was rolled back because $Orphan(\eta(s), f_i^m)$. However, $V_{\sigma(s)}(s) \not\prec tk(i, m).ts$, so $p_{\rho(s)}$ will request retransmission before the occurrence of the rb event, and $w1_{\rho(s)}(i, m) \Rightarrow \eta(s)$ (Rule CRB2.4). The final possibility is that $\eta(s)$ occurs after the wave but is discarded upon arrival. However, Rule CRB2.9 specifies that $\eta(s)$ is accepted if $V_{\sigma(s)}(s) \not\prec tk(i, m).ts$.

Case 2: $w1_{\sigma(s)}(i, m) \Rightarrow s$. If $\sigma(s) \neq i$ then $c1_i(i, m) \parallel s$, or $c1_i(i, m) \Rightarrow s$. Therefore, $V_{\sigma(s)}(s) > tk(i, m).Ltime$, or $V_{\sigma(s)}(s) \parallel tk(i, m).Ltime$. If $\sigma(s) = i$ then $c1_i(i, m) \Rightarrow s$, and $V_i(s) > tk(i, m).Ltime$. In either case CRB2.9 specifies that the message be accepted. ■

Theorem 21 *The completion of a valid wave in the Causal Recovery Protocol: Two Wave - Serial Failure satisfies RB2.*

Proof: Lemma 55 showed that $\neg \text{Orphan}(w1_j(i, m), f_i^m)$ for all $p_j \in \Pi$. Lemma 57 showed that $\nexists \eta(s)$ such that $w1_j(i, m) \Rightarrow \eta(s)$, and $\text{Orphan}(\eta(s), f_i^m)$. Therefore, for all $w2_j(i, m) \in \text{FW}(i, m)$, $\neg \text{Orphan}(w2_j(i, m), f_i^m)$, thus satisfying RB2(a). Since $w1_j(i, m) \Rightarrow w2_j(i, m)$ for all $p_j \in \Pi$, and Lemmas 57 and 58 have shown that RB2(b) holds for all $w1_j(i, m)$, clearly RB2(b) holds for all $w2_j(i, m)$. ■

To complete our correctness arguments for this protocol, we must also show that a polling wave instigated by a restart event will complete. To do this we must show that $rs_i^m \rightsquigarrow c2_i(i, m)$. As a first step we argue that all events in the polling waves are stable.

Lemma 59 *If $e'_j \in \text{PW1}(i, m) \cup \text{PW2}(i, m)$ then there does not exist f_j^k such that $\text{Disc}(e'_j, rs_j^k)$.*

Proof: Assume there exists f_j^k such that $\text{Disc}(e'_j, rs_j^k)$. This implies $rs_j^k \not\Rightarrow e'_j$, and $e'_j \not\Rightarrow rs_j^k$.

Case 1 ($i \neq j$): In this case, $c1_j(i, m) \Rightarrow w1_j(i, m) \Rightarrow c2_j(i, m) \Rightarrow w2_j(i, m)$. Given the restrictions on failure, $w2_j(i, m) \Rightarrow f_j^k$, or $w2_i(j, k) \Rightarrow f_i^m$. Rule CRB2.11 requires $c2_j(i, m)$ be logged before the occurrence of $w2_j(i, m)$ and that no event occur between $c2_j(i, m)$ and $w2_j(i, m)$. Therefore, $w2_j(i, m) \Rightarrow f_j^k$ implies $w2_j(i, m) = \text{LastEvent}(f_j^k)$, or $w2_j(i, m) \Rightarrow \text{LastEvent}(f_j^k)$. This in turn implies $c1_j(i, m) \Rightarrow w1_j(i, m) \Rightarrow c2_j(i, m) \Rightarrow w2_j(i, m) \Rightarrow rs_j^k$. Since e'_j must be one of these events, $e'_j \Rightarrow rs_j^k$. A similar argument applies if $w2_i(j, k) \Rightarrow f_i^m$. Rule CRB2.11 will insure $w2_i(j, k) \Rightarrow rs_i^m$. $rs_i^m \Rightarrow w2_i(j, k) \Rightarrow rs_i^m \Rightarrow c1_j(i, m) \Rightarrow w1_j(i, m) \Rightarrow c2_j(i, m) \Rightarrow w2_j(i, m)$. Therefore, $rs_j^k \Rightarrow e'_j$.

Case 2 ($i = j$): In this case $w1_i(i, m) \Rightarrow c1_i(i, m) \Rightarrow w2_i(i, m) \Rightarrow c2_i(i, m)$. Given the restrictions on failure, $c2_i(i, m) \Rightarrow f_j^k$, or $c2_j(j, m) \Rightarrow f_i^m$. First consider $c2_i(i, m) \Rightarrow f_j^k$. Rule CRB2.12 specifies $\text{Logged}(c2_i(i, m))$. Therefore, $c2_i(i, m) \Rightarrow f_j^k$ implies $c2_i(i, m) \Rightarrow$

rs_j^k , and $e'_j \Rightarrow rs_j^k$. In the case that $c2_j(j, m) \Rightarrow f_i^m$, $Logged(c2_j(j, m))$ implies $rs_j^k \Rightarrow c2_j(j, m) \Rightarrow rs_i^m \Rightarrow w1_i(i, m)$. Therefore, $rs_j^k \Rightarrow e'_j$. ■

Lemma 60 *If $e'_j \in PW1(i, m) \cup PW2(i, m)$ then there does not exist f_a^k or rb'_j such that rb'_j is instigated by rs_a^k , and $Disc(e'_j, rb'_j)$.*

Proof: Assume the contrary. Case 1 ($i \neq j$): According to Rule CRB2.4, e'_j will be eliminated if and only if $V_j(e'_j) > tk(a, k).ts$. The failure restrictions specify that $c2_j(i, m) \Rightarrow c1_j(a, k)$. Hence, $e'_j \Rightarrow c1_j(a, k)$. This implies $Orphan(e'_j, f_a^k)$ (Lemma 54). This contradicts the serial failure restrictions that specify that no polling wave event of the recovery from one failure may be orphaned by another failure.

Case 2 ($i = j$): The argument is similar to the case $i \neq j$. ■

Theorem 22 $rs_i^m \leadsto c2_i(i, m)$ in the Causal Recovery Protocol - Two Waves.

Proof: Lemmas 59 and 60 showed that $Stable(e'_j)$ for all $e'_j \in PW1(i, m) \cup PW2(i, m)$. According to Rule CRB2.2, $w1_i(i, m)$ will occur following rs_i^m . Since $w1_i(i, m)$ is stable, $rs_i^m \leadsto w1_i(i, m)$. Rules CRB2.3 and CRB2.5 specify that $w1_j(i, m), i \neq j$, occurs following $c1_j(i, m)$. Given that $w1_j(i, m)$ is stable for all $p_j \in \Pi$, $c1_j(i, m) \leadsto w1_j(i, m)$. Rule CRB2.6 will guarantee that $c1_i(i, m) \leadsto w2_i(i, m)$. Rule CRB2.11 guarantees that $w2_j(i, m), i \neq j$, occurs following $c2_j(i, m)$, and since $w2_j(i, m)$ is stable, that $c2_j(i, m) \leadsto w2_j(i, m)$. Given reliable communication, a token message originating in an event $w1_j(i, m)$ will arrive at $p_{j+1(modN)}$. The restrictions on failure guarantee that the message is not lost, however, Rules CRB2.8, CRB2.9, and CRB2.10 restrict the occurrence of incoming messages of the underlying computation. Polling messages are not considered part of the underlying computation, so these restrictions will not prevent their acceptance. In any case, since $w2_i(a, k) \Rightarrow rs_i^m$, for any failure f_a^k occurring before f_i^m , these restrictions would be met, and the receipt of a polling message would always be accepted. Thus, $c1_{j+1(modN)}(i, m)$ will occur following $w1_j(i, m)$, and $w1_j(i, m) \leadsto c1_{j+1(modN)}(i, m)$. A similar argument

can be made that $w2_j(i, m) \rightsquigarrow c2_{j+1(mod N)}(i, m)$. Because the token travels in a logical ring, $w1_i(i, m) \rightsquigarrow c1_i(i, m)$. We also know that $c1_i(i, m) \rightsquigarrow w2_i(i, m)$, and $w2_i(i, m) \rightsquigarrow c2_{j+1(mod N)}(i, m)$. Therefore, $w1_i(i, m) \rightsquigarrow c2_i(i, m)$, and $rs_i^m \rightsquigarrow c2_i(i, m)$. ■

4.7 Concurrent and Multiple Failures

In this section we will relax the restrictions on failure that we imposed in the previous section and show how our protocol must be modified to accommodate multiple and concurrent failures.

Failure during recovery affects not only the protocol design, but also the meaning of the protocol specifications. Several of the rules specifying CRB2 state that if one event occurs then another event must occur. For example, CRB2.2 specifies that if rs_i^m occurs then the token departure event $w1_i(i, m)$ must also occur, and $rs_i^m \Rightarrow w1_i(i, m)$. The semantics of such a specification are obvious when it is known that p_i will not fail between the occurrence of rs_i^m and $w1_i(i, m)$. If it is possible for p_i to fail at any time, then implicit in this specification is that the state information necessary to cause the $w1_i(i, m)$ event must survive the failure. Therefore, rs_i^m must be logged to stable storage so that it cannot be lost due to any failure that could occur before $w1_i(i, m)$. Otherwise there is no assurance that $rs_i^m \Rightarrow w1_i(i, m)$.

When multiple failures can occur there is also the potential for a process to be failed when a recovery token arrives, or to fail between the token's arrival and its processing. The protocol must be protected against such a loss of the token. For this reason, rules requiring that $w1_j(i, m) \rightsquigarrow c1_{j+1(mod N)}(i, m)$, and $w2_j(i, m) \rightsquigarrow c2_{j+1(mod N)}(i, m)$ are added to the protocol specifications. Implementation of these specifications will require action on the part of the processes that are not spelled out in the protocol rules. The process transmitting the token must guarantee that the token arrives and is processed. This may require a process to save the token information in stable storage before transmitting the token and to retransmit

the token until an acknowledgement is received from the receiving process.

Multiple failures can also give rise to two or more instantiations of the recovery protocol proceeding concurrently. When this occurs the waves of the concurrent recovery protocols may cross, so that, $w1_i(i, m) \Rightarrow c1_i(j, k) \Rightarrow c2_i(i, m)$, and $w1_j(j, k) \Rightarrow c1_j(i, m) \Rightarrow c2_j(j, k)$.

Concurrent failures will cause the the protocol presented in Section 4.6 not to work properly. One difficulty arises if a restart event in one process is orphaned by a failure in another process, i.e., there exist i, j, m, k such that $Orphan(rs_i^m, f_j^k)$. In this case, rs_i^m and the polling events instigated by this restart event will be eliminated during the rollback process. This problem is easily dealt with by treating the polling events as special messages forcing the propagation of the token and restoring the polling events after rollback has completed. These changes (outlined below) enable the protocol to handle concurrent failures but at a price of eliminating some non-orphan events.

During the first wave of the protocol the relationship between vector time and the partial order is in a state of flux. The isomorphism of vector time to the partial order holds for those events that occur after a wave event, $w1_j(i, m) \in PW1(i, m)$. The rules of the protocol specify that inconsistencies in the partial order are removed before the rollback event instigated by $w1_j(i, m)$ occurs. This is not true for those events ahead of the wave. This is not a problem when a wave can be completed without disruption of any of the causal relationships that existed at the time of failure.

The rollback wave of one process can disrupt the partial order ahead of the wave instigated by another process. This occurs if there is a causal relationship between lost or orphan events in one failed process and orphan or lost events in another failed process. As a result, events occurring between the concurrent waves may have timestamps that indicate that they are orphans, when in fact they are not. Consequently it is possible to identify all the orphan events by their timestamps, but it is not possible to guarantee that all the events so identified are orphans.

Figure 4.17 shows how this can occur (the checkpoints and rollback events associated

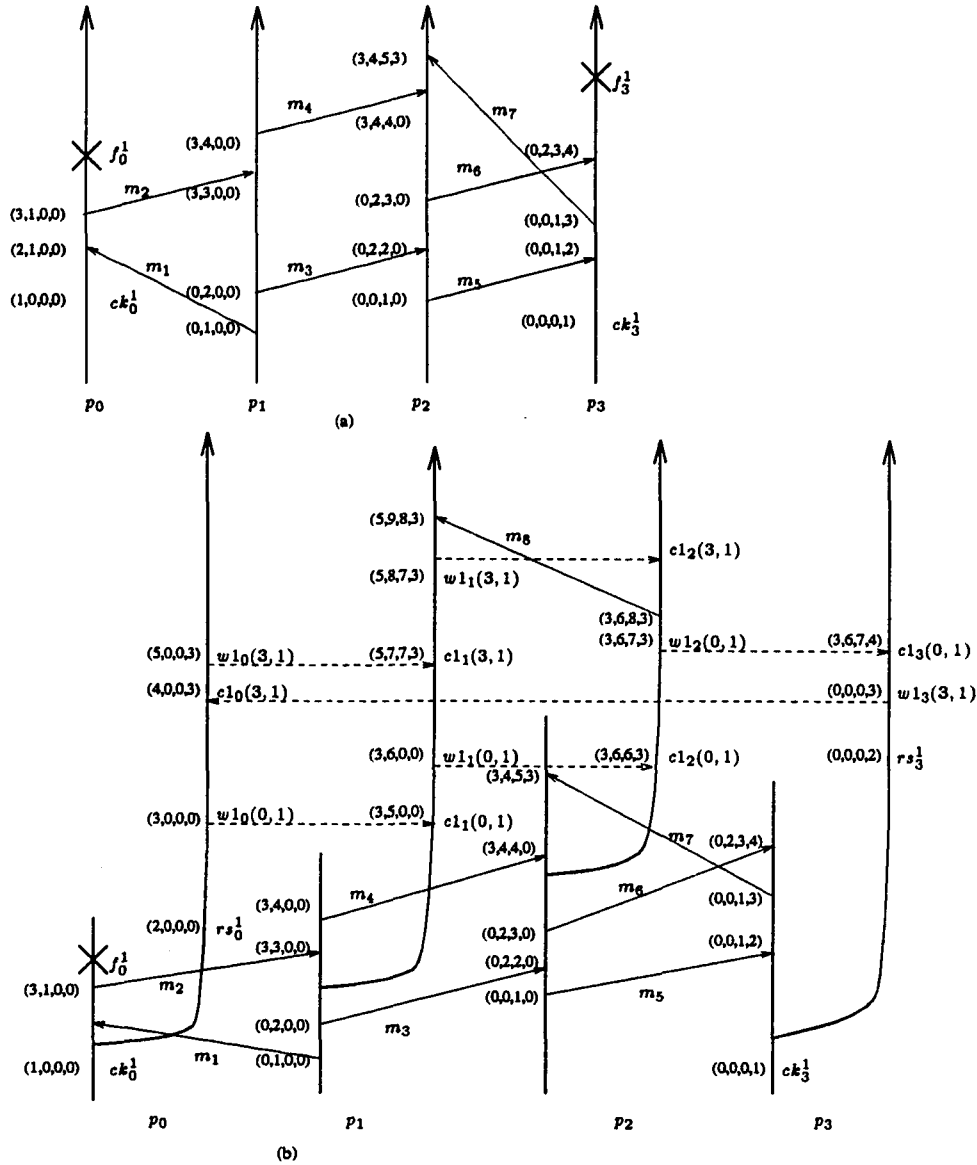


Figure 4.17: Concurrent Failures

with the token have been left out for clarity). In part(a) of the figure both p_0 and p_3 are recovering from failure. The token from p_0 has been propagated to p_3 . The token from p_3 has reached p_0 . Part (b) of the figure shows the state of the system when the token from p_3 arrives at p_2 . Message m_7 from p_3 was eliminated when p_2 rolled back. This eliminated

any events in p_2 that were orphaned by p_3 's failure. However, the timestamp of message m_8 makes it appear to be an orphan of p_3 's failure. This occurs because there was a causal relationship between the lost events in p_0 and the event orphaned by p_3 . The receipt of m_8 will be eliminated when p_2 rolls back even though m_8 does not originate in an orphan event.

Since our protocol depends on timestamps to identify orphan events it will in some instances eliminate events occurring between the waves that are not orphans. Therefore, the protocol we present in this section can not guarantee that all non-orphan events are restored. It can guarantee that the system is returned to a consistent state and that messages originating in send events not eliminated from the partial order by rollback are received, so it meets the weaker rollback conditions specified by RB.

Three changes need to be made to the protocol to make it accommodate concurrent failure. First, during rollback a process will not request retransmission of messages that originated in non-orphaned events. It will simply reinstate them from the message log. This requires a modification of the *Retransmit* rules.

Retransmit($tk.ts, tk.id, tk.seq, id, c.event$)

For all $e'_{id} = s$ such that:

$$s \Rightarrow c.event \wedge$$

$$V_i^i(s) > tk.seq(i) \wedge$$

$$V_i(s) \not\geq tk.ts$$

s is retransmitted to $p_{tk.id}$

The second change requires that all polling events are also reinstated from the message log before the rollback event occurs. In the previous protocol only those messages whose timestamps were not greater than the token's timestamp were replayed from the message log during the rollback procedure. In this protocol all polling events are replayed from the log regardless of their timestamp. In this way polling events will never be eliminated from system execution by the rollback protocol.

Finally, the token message must also be treated specially by the process propagating it and by the receiving process. The process which sends the token must make sure that the token is received. This may require repeated transmissions and acknowledgements to insure that the token arrival event occurs. The token message must always be received regardless of its timestamp. A recovering process will receive an incoming token, save the token information to stable storage, rollback if necessary and propagate the token even though it is nominally frozen until its own token returns.

4.7.1 Formal Specification

Causal Recovery Protocol - Concurrent Failures

CRB3.1 The occurrence of rs_i^m implies

$$\begin{aligned} &LastEvent(f_i^m) \Rightarrow rs_i^m \wedge \\ &tk(i, m).ts = V_i(rs_i^m) \wedge \\ &tk(i, m).id = i. \end{aligned}$$

A restart event occurs when the latest event that occurred prior to failure is recovered from stable storage. A token incorporating the timestamp of the restart event and the id of the recovering process is created during this event.

CRB3.2. $w1_i(i, m)$ occurs iff

$$\begin{aligned} & \exists rs_i^m \text{ such that } rs_i^m \Rightarrow w1_i(i, m) \wedge \\ & \exists ck_i' \text{ such that } ck_i' \Rightarrow w1_i(i, m) \wedge CK(ck_i', rs_i^m) \wedge \\ & \left[\begin{array}{l} \nexists e_i' \text{ such that } rs_i^m \Rightarrow e_i' \Rightarrow w1_i(i, m) \wedge \\ e_i' \text{ is an event of the underlying computation.} \end{array} \right] \end{aligned}$$

A formerly failed process creates and propagates a token, event $w1_i(i, m)$, immediately after the occurrence of a restart event rs_i^m .

CRB3.3 $w1_i(i, m) \leadsto c1_{i+1(\text{mod}N)}(i, m)$.

It is the responsibility of the i^{th} process to guarantee arrival of token at $p_{i+1(\text{mod}N)}$.

CRB3.4. A rollback event, rb_j' , is instigated by rs_i^m iff

$$\begin{aligned} & c1_j(i, m) \Rightarrow rb_j' \wedge \\ & \left[\begin{array}{l} \nexists e_j' \text{ such that } c1_j(i, m) \Rightarrow e_j' \Rightarrow rb_j' \wedge \\ e_j' \text{ is an event of the underlying computation.} \end{array} \right] \end{aligned}$$

Rollback is immediately instigated by the arrival of the token.

CRB3.5. The occurrence of rb_j' instigated by rs_i^m implies

$$\begin{aligned} e_j' \Rightarrow c1_j(i, m) \text{ iff } & \left[\begin{array}{l} V_j(e_j') \not\Rightarrow tk(i, m).ts \vee \\ e_j' = \eta(s) \wedge V_{\sigma(s)}(s) \not\Rightarrow tk(i, m).ts \vee \\ \exists k, l : e_j' \in PW1(k, l) \cup PW2(k, l) \end{array} \right] \wedge \\ & \text{Retransmit}(tk(i, m).ts, tk(i, m).id, tk(i, m).seq, j, c1_j(i, m)). \end{aligned}$$

A rollback event implies that all orphan events have been eliminated except those message receipts that do not originate in an orphan event. It also implies that the necessary messages have been retransmitted.

CRB3.6. $w1_j(i, m), i \neq j$ occurs iff

$$\begin{aligned} & \exists rb_j' \text{ instigated by } rs_i^m \text{ such that } rb_j' \Rightarrow w1_j(i, m) \wedge \\ & \exists ck_j' \text{ such that } ck_j' \Rightarrow w1_j(i, m) \wedge CK(ck_j', rb_j') \wedge \\ & \left[\begin{array}{l} \nexists e_j' \text{ such that } rb_j' \Rightarrow e_j' \Rightarrow w1_j(i, m) \wedge \\ e_j' \text{ is an event of the underlying computation.} \end{array} \right] \end{aligned}$$

A non-failed process will propagate the token only after it has rolled back.

CRB3.7. For $i \neq j$, $w1_j(i, m) \leadsto c1_{j+1(\text{mod}N)}(i, m)$.

CRB3.8. $c1_i(i, m) \Rightarrow w2_i(i, m)$.

The second polling wave begins when the first wave is completed.

CRB3.9. The occurrence of $w2_i(i, m)$ implies

$$\begin{aligned} tk(i, m).Ltime &= V_i(c1_i(i, m)) \wedge \\ w2_i(i, m) &\leadsto c2_{i+1(mod N)}(i, m). \end{aligned}$$

The vector time of the last event of the first polling wave is put in the token before the beginning of the second polling wave.

CRB3.10. The occurrence of $\eta(s)$ where $\rho(s) = p_i$, and $rs_i^m \Rightarrow \eta(s)$, implies $c1_i(i, m) \Rightarrow \eta(s)$.

A recovering process will not accept any incoming messages until the first polling wave is completed.

CRB3.11 The occurrence of $\eta(s)$ where $w1_{\rho(s)}(i, m) \Rightarrow \eta(s)$ implies that

$$\begin{aligned} V_{\sigma(s)}(s) &\not\geq tk(i, m).ts \vee \\ V_{\sigma(s)}(s) &> tk(i, m).Ltime \vee \\ V_{\sigma(s)}(s) &\parallel tk(i, m).Ltime \wedge \sigma(s) \neq i. \end{aligned}$$

Any message arriving after a polling wave must be compared to the token timestamp and the timestamp of $c1_i(i, m)$ to determine whether it originated in an orphan event.

CRB3.12 The occurrence of s such that $w1_{\sigma(s)}(i, m) \Rightarrow s \Rightarrow c2_{\sigma(s)}(i, m)$ implies $c1_{\rho(s)}(i, m) \Rightarrow \eta(s)$.

Messages are not allowed to cross the first polling wave.

CRB3.13 $w2_j(i, m), i \neq j$ occurs iff

$$\begin{aligned} &c2_j(i, m) \Rightarrow w2_j(i, m) \wedge \\ &\left[\begin{array}{l} \exists e'_j \text{ such that } c2_j(i, m) \Rightarrow e'_j \Rightarrow w2_j(i, m) \wedge \\ e'_j \text{ is an event of the underlying computation.} \end{array} \right] \end{aligned}$$

CRB3.14. For $i \neq j$, $w2_j(i, m) \leadsto c2_{j+1(mod N)}(i, m)$.

CRB3.15 The occurrence of $e'_j = c1_j(i, m)$, or $e'_j = c2_j(i, m)$, implies $Logged(e'_j)$.

4.7.2 Correctness

First, we will show that the polling events are stable. Showing that these events persist though failure and rollback is necessary in later arguments about the protocol's operation.

Lemma 61 *If $e'_j \in PW1(i, m) \cup PW2(i, m)$ then there does not exist f_j^k such that $e'_j \Rightarrow f_j^m \wedge Disc(e'_j, rs_j^k)$*

Proof: Assume there exists f_j^k such that $Disc(e'_j, rs_j^k)$. This implies $rs_j^k \not\Rightarrow e'_j$ and $e'_j \not\Rightarrow rs_j^k$.

Rule CRB3.15 specifies that the occurrence of $c1_j(i, m)$ implies $Logged(c1_j(i, m))$. Therefore, $c1_j(i, m) \Rightarrow LastEvent(f_j^k)$, or $c1_j(i, m) = LastEvent(f_j^k)$, and $c1_j(i, m) \Rightarrow rs_j^k$. Similarly, Rule CRB3.15 specifies $Logged(c2_j(i, m))$, so $e'_j = c2_j(i, m)$, or $e'_j = c1_j(i, m)$ implies $e'_j \Rightarrow rs_j^k$.

Now consider the case where $e'_j = w1_j(i, m)$, or $e'_j = w2_j(i, m)$, and $i \neq j$. Rule CRB3.6 requires that there exist a checkpoint ck'_j , such that $CK(ck'_j, rb'_j)$, where rb'_j is the rollback event instigated by $c1_j(i, m)$. Since CRB3.6 also specifies that no event of the underlying computation occur between rb'_j and $w1_j(i, m)$, $w1_j(i, m)$ can always be recovered, and $w1_j(i, m) \Rightarrow LastEvent(f_j^k)$, or $w1_j(i, m) = LastEvent(f_j^k)$. Therefore, $e'_j = w1_j(i, m)$ implies $e'_j \Rightarrow rs_j^k$. A similar argument can be made in the case where $e'_j = w2_j(i, m)$. $Logged(c2_j(i, m))$, and no receive event of the underlying computation may occur between $c2_j(i, m)$ and $w2_j(i, m)$, therefore, $w2_j(i, m) \Rightarrow rs_j^k$.

Finally consider the case that $e'_j = w1_i(i, m)$, or $e'_j = w2_i(i, m)$. Rule CRB3.2 specifies that a checkpoint of rs_i^m must occur before $w1_i(i, m)$ takes place. Therefore, $w1_i(i, m)$ can always be recovered, and $w1_i(i, m) \Rightarrow rs_j^k$. Rules CRB3.8 and CRB3.15 guarantee that $w2_i(i, m)$ is recoverable from the stable logs, and $w2_i(i, m) \Rightarrow rs_j^k$. ■

Lemma 62 *If $e'_j \in PW1(i, m) \cup PW2(i, m)$ then there does not exist f_a^k or rb'_j such that rb'_j is instigated by rs_a^k , and $Disc(e'_j, rb'_j)$.*

Proof: Assume the contrary, so that there exists a failure f_a^k and a rollback event rb'_j instigated by rs_a^k such that $Disc(e'_j, rb'_j)$. Rule CRB3.5 specifies that if $e'_j \in PW1(i, m) \cup PW2(i, m)$ then $e'_j \Rightarrow c1_j(i, m)$. Rule CRB3.4 specifies that $c1_j(i, m) \Rightarrow rb'_j$. Therefore, $e'_j \Rightarrow rb'_j$, and $\neg Disc(e'_j, rb'_j)$. ■

Lemmas 52 and 53 still hold in the modified protocol. However, we can no longer show Lemma 54.

Lemma 63 *For any $w1_j(i, m)$ event as it is specified in the Causal Recovery Protocol - Concurrent Failures, $\neg Orphan(w1_j(i, m), f_i^m)$.*

Proof: Assume the contrary, that there exists $w1_j(i, m)$ for which $Orphan(w1_j(i, m), f_i^m)$ is true. Let $w1_j(i, m)$ be the earliest token transmission event in the polling wave for which $Orphan$ is true. If that is the case, then there exists e'_i , such that $Disc(e'_i, rs_i^m)$, and $e'_i \Rightarrow w1_j(i, m)$. Then there must also exist e'_j , such that $e'_i \Rightarrow e'_j \Rightarrow w1_j(i, m)$. Let e'_j be the earliest such event in p_j . It is not possible for $e'_j = c1_j(i, m)$, because this would imply $Orphan(w1_{j-1(modN)}(i, m), f_i^m)$ contradicting the assumption that $w1_j(i, m)$ was the earliest token transmission which was an orphan. Therefore, $e'_j \Rightarrow c1_j(i, m)$. According to Lemma 53, $Orphan(e'_j, f_i^m)$ implies $V_j(e'_j) > tk(i, m).ts$. Rules CRB3.4 and CRB3.6 of the protocol specify that a rollback event rb'_j occurs such that $c1_j(i, m) \Rightarrow rb'_j \Rightarrow w1_j(i, m)$. Therefore, $e'_j \Rightarrow rb'_j$. However, if $e'_j \Rightarrow rb'_j$, and $V_j(e'_j) > tk(i, m).ts$ then e'_j is a polling event, or e'_j is a receive event $\eta(s)$ such that $V_{\sigma(s)} \not> tk(i, m).ts$.

If e'_j is a polling event and is the earliest event in p_j such that $Orphan(e'_j, f_i^m)$, then $\exists k, n$ such that $e'_j = c1_j(k, n)$, or $e'_j = c2_j(k, n)$. $\neg Orphan(w1_{j-1(modN)}(i, m), f_i^m)$, and $Orphan(c1_j(k, m), f_i^m)$ implies $j - 1(modN) = i$, and $Disc(w1_i(k, n), rs_i^m)$. However, we proved in Lemma 61 that $\neg Disc(w1_i(k, n), rs_i^m)$ for all f_i^m . The other possibility is that $e'_j = c2_j(k, n)$ for some k and n . Once again this implies $Disc(w2_i(k, n), rs_i^m)$, which contradicts the results of Lemma 61. Therefore, if $e'_j = c1_j(k, n)$, or $e'_j = c2_j(k, n)$, then $\neg Orphan(e'_j, f_i^m)$.

The other possibility is that e'_j is a receive event $\eta(s)$ such that $V_{\sigma(s)} \not> tk(i, m).ts$. In this case $\neg Orphan(s, f_i^m)$. This implies there does not exist e'_i such that $e'_i \Rightarrow s$, and $Disc(e'_i, rs_i^m)$ (Lemma 53). Therefore, $\neg Orphan(\eta(s), f_i^m)$. ■

Lemmas 56 and 57 still hold in the modified protocol. Lemma 64 will show that RB(c) holds for **Causal Recovery Protocol - Concurrent Failures**.

Lemma 64 *If $\neg \text{Disc}(s, w1_{\sigma(s)}(i, m))$ then $\eta(s) \Rightarrow w1_{\rho(s)}(i, m) \vee w1_{\rho(s)}(i, m) \Rightarrow \eta(s)$.*

Proof: Case 1: $s \Rightarrow w1_{\sigma(s)}(i, m)$. This implies $V_{\sigma(s)}(s) \not\prec tk(i, m).ts$. If the receipt of s is lost by a failed process then $tk(i, m).seq(\sigma(s)) < V_{\sigma(s)}^{\sigma(s)}$. Therefore, s will be retransmitted during the rollback process (Rule CRB3.5). If $\eta(s)$ arrives after the token it will be accepted, because the first disjunct of CRB3.11 will be satisfied. If $\eta(s)$ occurs before the arrival of the token, $\eta(s) \Rightarrow w1_{\rho(s)}(i, m)$ (Rule CRB3.5).

Case 2: $w1_{\sigma(s)}(i, m) \Rightarrow s$. In this case $V_{\sigma(s)}(s) \parallel tk(i, m).Ltime$, or $V_{\sigma(s)}(s) > tk(i, m).Ltime$ and $\sigma(s) = i$. Therefore, $\eta(s)$ will be accepted, and $w1_{\rho(s)}(i, m) \Rightarrow \eta(s)$. ■

Theorem 23 *The completion of a valid wave in the Causal Recovery Protocol- Concurrent Failures satisfies RB.*

Proof: Lemma 63 showed that $\neg \text{Orphan}(w1_j(i, m), f_i^m)$ for all $p_j \in \Pi$. Lemma 57 showed that $\nexists \eta(s)$ such that $w1_j(i, m) \Rightarrow \eta(s)$ and $\text{Orphan}(\eta(s), f_i^m)$. Therefore, for all $w2_j(i, m) \in FW(i, m)$, $\neg \text{Orphan}(w2_j(i, m), f_i^m)$, thus satisfying RB(a). Since $w1_j(i, m) \Rightarrow w2_j(i, m)$ for all $p_j \in \Pi$, and Lemmas 57 and 64 have shown that RB(b) and RB(c) hold for all $w1_j(i, m)$, clearly RB(b) and RB(c) hold for all $w2_j(i, m)$. ■

Theorem 24 *$rs_i^m \rightsquigarrow c2_i(i, m)$ in the Causal Recovery Protocol - Concurrent Failures.*

Proof: Lemmas 61 and 62 showed that $\text{Stable}(e'_j)$ for all $e'_j \in PW1(i, m) \cup PW2(i, m)$. According to Rule CRB3.2, $w1_i(i, m)$ will occur following rs_i^m . Since $w1_i(i, m)$ is stable, $rs_i^m \rightsquigarrow w1_i(i, m)$. Rule CRB3.3 specifies $w1_i(i, m) \rightsquigarrow c1_{i+1(mod N)}(i, m)$ Rules CRB3.4 and CRB3.6 specify that $w1_j(i, m), i \neq j$, occurs following $c1_j(i, m)$. Given that $w1_j(i, m)$ is stable for all $p_j \in \Pi$, $c1_j(i, m) \rightsquigarrow w1_j(i, m), i \neq j$. Rule CRB3.7 implies $w1_j(i, m) \rightsquigarrow c1_{j+1(mod N)}(i, m), i \neq j$. Because the token travels in a logical ring, $w1_i(i, m) \rightsquigarrow c1_i(i, m)$.

Rules CRB3.8 and CRB3.9 guarantee that $c1_i(i, m) \rightsquigarrow w2_i(i, m) \rightsquigarrow c2_{i+1(mod N)}(i, m)$. Rule CRB3.13 guarantees that $w2_j(i, m), i \neq j$, occurs following $c2_j(i, m)$, and since $w2_j(i, m)$ is stable, that $c2_j(i, m) \rightsquigarrow w2_j(i, m), i \neq j$. Rule CRB3.14 specifies $w2_j(i, m) \rightsquigarrow c2_{j+1(mod N)}(i, m)$. Therefore, $c1_i(i, m) \rightsquigarrow c2_i(i, m)$, $w1_i(i, m) \rightsquigarrow c2_i(i, m)$, and $rs_i^m \rightsquigarrow c2_i(i, m)$. ■

Chapter 5

Conclusions and Future Research

The basic premise that originally motivated this research was that the function of vector clocks and synchronized clocks was similar enough so that vector timestamps could be mechanically substituted for real timestamps in certain synchronous protocols. In cases where this was possible, and we hoped that this would be true for a large class of distributed problems, vector clocks could be used to implement distributed protocols which were designed as if synchronized clocks were available. This would have the effect of simplifying protocol design without imposing the performance demands of clock synchronization on the system. Unfortunately, upon investigation we found that while the temporal order and causal order share similar characteristics, the lack of isomorphism between the two orderings prevents the direct substitution of vector timestamps for real timestamps.

Depending on the nature of the distributed problem either the temporal order or causal order is more relevant to the problem's solution. In the cases of distributed termination detection and deadlock detection, knowledge of the temporal order led to straightforward solutions. In both cases identification of a latest event, an idle event in the case of termination detection, a request event in the case of deadlock detection, was integral to the synchronous protocols presented. This identification is possible because the order imposed by synchronized clocks is a total order. Because the causal order is a partial order, identifi-

cation of a unique latest event may not be possible in a causally synchronous system which uses vector clocks. As a result, vector timestamps can not be mechanically substituted for real timestamps, and the synchronous deadlock detection and termination detection protocols must be modified to accommodate the inability of vector time to impose a total order on events.

However, the necessary protocol modifications were not extensive, and much of the simple structure and performance efficiency of the synchronous protocols could be preserved in the causally synchronous protocols. Because the functioning of the synchronous and causally synchronous protocols was similar, the correctness arguments used in the synchronous system environment could also be used with slight modification to argue that the causally synchronous protocols were correct.

The problem of optimistic recovery differs from distributed termination and deadlock detection in that the causal order is more useful than the temporal order in design of an optimistic recovery protocol. The causal relationship between events determines what actions should be taken to restore a system to consistency. Real timestamps can be used to show that no causal relationship exists between a set of events, but they can't be used to show that such a relationship does exist. Therefore, the temporal order does not supply the needed information, and the synchronous optimistic recovery protocol we presented is not as logical or efficient as the optimistic recovery protocols which use vector clocks.

The FCFS centralized service problem we presented in Chapter 1 is another example of a problem where the causal order is more relevant than the temporal order. In this case vector clocks, because their ordering is isomorphic to the causal order, can be used more readily than synchronized clocks in design of a solution.

Even though we discovered that vector time could not be directly substituted for real time in distributed protocols, we found that solving a problem first in a synchronous environment led us to a useful solution in a causally synchronous environment. Availability of synchronized clocks and knowledge of the temporal order provides insight into the prop-

erties of a distributed computation that is not available in an asynchronous system. This insight often leads to an obvious and simple solution. Rana's termination detection protocol is a good example of how imposing order on the events of a distributed computation simplifies the problem and its solution. Consideration of order is also useful in understanding and solving the distributed deadlock problem. Existing protocols for this problem were either incorrect or inefficient, primarily because there was no consideration of event order in these solutions. Identifying the order in which events of the underlying computation occurred made the solution to the deadlock problem straightforward. The similarities between the causal order and temporal order are strong enough that these synchronous protocols provided a template for our causal protocols.

Knowledge of event order, even if it is only knowledge of the causal partial order, is a powerful tool. With it we were able to develop a methodology for analysis and design of distributed protocols. Using the causal order we can specify conditions which must be met for a protocol to be correct, define the axiomatic protocol specifications, and structure the reasoning about the correctness of the specified protocol. The advantage of using causality as a unifying concept is that the correctness conditions and protocol specifications are defined in terms of local states and local events. This means that we need consider only local process state when arguing that a causal protocol is correct.

Whereas our emphasis is on event order and local state, the prevailing framework for presentation and analysis of distributed protocols emphasizes global state and ignores event order. This is a natural consequence of the fact that in an asynchronous distributed system without logical clocks there is no way to explicitly identify the event order. Without knowledge of event order it is difficult to reason about the operation of a distributed computation using only local state, therefore, it is necessary to try to reason about global state.

The standard technique used to formally argue that a distributed protocol is correct is to define a global invariant and attempt to show that the protocol preserves the invariant. Both Francez and Dijkstra[14, 15] propose global invariants for justification of their termination

detection protocols. Sistla and Welch[47] use global arguments in presentation of their optimistic recovery protocol.

The difficulty with global invariants and global reasoning is that it is easy to define a global invariant, but it is hard to show that the invariant is satisfied. To show that a global invariant is satisfied in every instance, all possible global states must be identified and considered. This is a daunting task when the distributed computation being analyzed is complex.

A good example of the difficulties that can arise can be seen in [49]. Kshemkalyani and Singhal identify errors in Choudary, et.al.'s[8], priority-probe deadlock detection protocol that we outlined in Chapter 3 and propose necessary corrections. To show that their solution is correct, they define two global invariants. Paraphrasing, the invariants are:

- For all T_i and T_j , such that T_i waits for T_j , there exists n such that a probe initiated by T_i will become a member of $Probe_Q_j$ in n steps (T_i and T_j are transactions).
- For all T_i and T_j , a probe initiated by T_i in $Probe_Q_j$ implies T_i waits for T_j .

Clearly these invariants, if always satisfied, ensure detection of deadlock and prevent detection of false deadlock. The problem is showing that every action of the protocol preserves the invariant. Kshemkalyani and Singhal purport to show that their protocol does. They do this by trying to identify all relevant execution threads of their protocol, and then arguing that in no case are the invariants violated. We won't repeat the arguments of Chapter 3 here, but while Choudary, et.al., and Natarajan did not state these invariants formally, it is obvious they thought their versions of priority-probe deadlock detection satisfied them. They even posed arguments to that effect. However, it has been shown that these invariants are not satisfied by their protocols. When making their arguments they failed to identify a possible execution sequence that violated one of the invariants. In arguing that their protocol is correct, Kshemkalyani and Singhal try to identify every relevant execution path. However, how do they know that they have identified every path? How can they prove that

they have? That is the crux of the difficulty with global invariants and global reasoning. In a complicated system it is hard to know that every possibility has been considered.

Global invariants have strong appeal because they are consistent with the single processor model, the schema most of us are comfortable with and find most logical. The problem is that a global invariant perpetuates the illusion of global state. Since there is no global state in an asynchronous distributed system, the use of global invariants in an asynchronous system is not a productive way to view such a system.

We believe we have shown that causal reasoning, by concentrating on local state, provides a superior framework for analyzing distributed problems. The polling model can be used to structure and evaluate distributed protocols. Correctness arguments can be stated purely in terms of the local state of a process at an event or set of events. The advantage of this is that there is no need to try to identify all possible paths of execution. It is only necessary to argue about what is true or false about a particular event using the causal order. Admittedly, the causal correctness criteria are not as intuitively obvious as a global invariant, but once they are designed it is much easier to show that they are correct, as the protocols we have presented have demonstrated.

In the future we plan to apply the causal order and the polling wave model to other distributed problems. The resource deadlock problem we analyzed in Chapter 3 is only one variant of the distributed deadlock problem. Other variants include communication deadlock and multiple resource deadlock. We also plan to apply this methodology to distributed election, garbage collection and agreement protocols. Even if the use of vector time does not result in more efficient protocols, we think the polling model will provide a useful framework for analyzing these problems.

Bibliography

- [1] G. Ricart and A. Agrawala, "An optimal algorithm for mutual exclusion in computer networks," *Communications ACM*, vol. 24, no. 1, pp. 9–17, 1981.
- [2] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications ACM*, vol. 21, no. 7, pp. 558–565, 1978.
- [3] I. Suzuki and T. Kasami, "A distributed mutual exclusion algorithm," *ACM Trans. Computer Systems*, vol. 3, no. 4, pp. 344–349, 1985.
- [4] M. Maekawa, "A \sqrt{N} algorithm for mutual exclusion in decentralized systems," *tocs*, vol. 3, no. 2, pp. 145–159, 1985.
- [5] K. Chandy, J. Misra, and L. Haas, "Distributed deadlock detection," *ACM Trans. Computer Systems*, vol. 1, no. 2, pp. 144–156, 1983.
- [6] S. Huang, "A distributed deadlock detection algorithm for csp-like communication," *ACM Trans. Programming Languages and Systems*, vol. 12, no. 1, pp. 102–122, 1990.
- [7] A. Elmagarmid, N. Soundararajan, and M. Liu, "A distributed deadlock detection and resolution algorithm and its correctness proof," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1443–1452, 1988.
- [8] A. Choudhary, W. Kohler, J. Stankovic, and D. Towsley, "A modified priority based probe algorithm for distributed deadlock detection and resolution," *IEEE Trans. Software Engineering*, vol. 15, no. 1, pp. 10–17, 1989.
- [9] D. Badal, "The distributed deadlock detection algorithm," *ACM Trans. Computer Systems*, vol. 4, no. 4, pp. 320–337, 1986.
- [10] V. Gligor and S. Shattuck, "On deadlock detection in distributed systems," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 5, pp. 435–440, 1980.
- [11] M. Sinha and N. Natarajan, "A priority based distributed deadlock detection algorithm," *IEEE Trans. Software Engineering*, vol. SE-11, no. 1, pp. 67–80, 1985.
- [12] D. Menasce and R. Muntz, "Locking and deadlock detection in distributed data bases," *IEEE Trans. Software Engineering*, vol. SE-5, no. 3, pp. 195–202, 1979.
- [13] N. Francez, "Distributed termination," *ACM Trans. Programming Languages and Systems*, vol. 2, no. 1, pp. 42–55, 1980.

- [14] N. Francez and M. Rodeh, "Achieving distributed termination without freezing," *IEEE Trans. Software Engineering*, vol. SE-8, no. 3, pp. 287-292, 1982.
- [15] E. Dijkstra, W. Feijen, and A. van Gasteren, "Derivation of a termination detection algorithm for distributed computations," *Inf. Process. Lett.*, vol. 16, no. 5, pp. 217-219, 1983.
- [16] S. Rana, "A distributed solution of the distributed termination problem," *Information Processing Letters*, vol. 17, pp. 43-46, 1983.
- [17] R. Topor, "Termination detection for distributed computations," *Inf. Process. Lett.*, vol. 18, no. 1, pp. 33-36, 1984.
- [18] K. Apt, "Correctness proofs of distributed termination algorithms," *ACM Trans. Programming Languages and Systems*, vol. 8, no. 3, pp. 388-405, 1986.
- [19] F. Mattern, "New algorithms for distributed termination detection in asynchronous message passing systems," Report 42/85, University of Kaiserslautern, 1985.
- [20] J. Misra, "Detecting termination of distributed computations using markers," in *Proceedings of the ACM SIGACT-SIGOPS Symposium of Principles of Distributed Computing*, pp. 290-294, 1983.
- [21] H. Abu-Amara, "Fault tolerance distributed algorithm for election in complete networks," *IEEE Trans. Computers*, vol. 37, no. 4, pp. 449-453, 1988.
- [22] H. Garcia-Molina, "Elections in a distributed computing system," *IEEE Trans. Computers*, vol. C-31, no. 1, pp. 48-59, 1982.
- [23] G. Peterson, "An $O(n \log n)$ unidirectional algorithm for the circular extrema problem," *ACM Trans. Programming Languages and Systems*, vol. 4, no. 4, pp. 758-762, 1982.
- [24] A. Itai, S. Kuten, Y. Wolfstahl, and S. Zaks, "Optimal distributed t -resilient election in complete networks," *IEEE Trans. Software Engineering*, vol. 16, no. 4, pp. 415-420, 1990.
- [25] G. Lelann, "Distributed systems—towards a formal approach," in *Proceedings of the IFIP Congress*, pp. 155-160, 1977.
- [26] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, vol. 27, no. 2, pp. 228-234, 1980.
- [27] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals problem," *ACM Trans. Programming Languages and Systems*, vol. 4, pp. 382-401, 1982.
- [28] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, pp. 374-382, 1985.
- [29] M. Fischer, N. Lynch, and M. Merritt, "Easy impossibility proofs for distributed consensus problems," *Distributed Computing*, vol. 1, no. 1, pp. 26-39, 1986.

- [30] T. Srikanth and S. Toueg, "Optimal clock synchronization," *J. ACM*, vol. 34, no. 3, pp. 627-645, 1987.
- [31] H. Kopetz and W. Ochsenreiter, "Clock synchronization in distributed real-time systems," *ACM Trans. Computer Systems*, vol. C-36, no. 8, pp. 933-940, 1987.
- [32] F. Mattern, "Virtual time and global states of distributed systems," in *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms* (M. C. et. al., ed.), pp. 215-226, Elsevier Science Publishers B. V., 1989.
- [33] J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," in *Proceedings 11th Australian Computer Science Conference*, pp. 56-66, 1988.
- [34] P. Kearns and B. Koodalatupuram, "Immediate ordered service in distributed systems," in *Proceedings of the Ninth International Conference on Distributed Computing Systems*, pp. 611-618, 1989.
- [35] C. Morgan, "Global and logical time in distributed algorithms," *Inf. Process. Lett.*, vol. 20, no. 4, pp. 189-194, 1985.
- [36] G. Neiger and S. Toueg, "Substituting for real time and common knowledge in asynchronous distributed systems," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pp. 281-293, 1987.
- [37] M. Ahamad, P. Hutto, and R. John, "Implementing and programming causal distributed shared memory," in *Proceedings of the Eleventh International Conference on Distributed Computing Systems*, pp. 274-281, 1991.
- [38] C. Hoare, "Communicating sequential processes," *Communications ACM*, vol. 21, no. 8, pp. 666-677, 1978.
- [39] S. S. Isloor and T. A. Marsland, "An effective 'on-line' deadlock detection technique for distributed data base management systems," in *Proceedings COMSAC 1978*, pp. 283-288, 1978.
- [40] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. Software Engineering*, vol. SE-13, no. 1, pp. 23-31, 1987.
- [41] P. Ramanathan and K. Shin, "Use of common time base for checkpointing and rollback recovery in a distributed system," *IEEE Trans. Software Engineering*, vol. 19, no. 6, pp. 571-583, 1993.
- [42] A. Borg, J. Baumbach, and S. Glazer, "A message system supporting fault tolerance," in *Proceedings of the Ninth ACM Symposium on Operating Systems*, pp. 90-99, 1983.
- [43] M. Powell and D. Presotto, "Publishing: A reliable broadcast communication mechanism," in *Proceedings of the Ninth ACM Symposium on Operating Systems*, pp. 100-109, 1983.

- [44] R. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Trans. Computer Systems*, vol. 3, no. 3, pp. 204–226, 1985.
- [45] K. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.
- [46] D. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and checkpointing," *Journal of Algorithms*, vol. 11, no. 3, pp. 462–491, 1990.
- [47] A. Sistla and J. Welch, "Efficient distributed recovery using message logging," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pp. 223–238, 1989.
- [48] E. Elnozahy and W. Zwaenepoel, "Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit," *ACM Trans. Computer Systems*, vol. 41, no. 5, pp. 526 – 531, 1992.
- [49] A. Kshemkalyani and M. Singhal, "Invariant-based verification of a distributed deadlock detection algorithm," *IEEE Trans. Software Engineering*, vol. 17, no. 8, pp. 789–799, 1991.

VITA

Sandra L. Peterson was born in Coral Gables, Florida, on June 19, 1952. A graduate of Melbourne High School in Melbourne, Florida, she received her B.A. in Mathematics from the College of William and Mary, Williamsburg, Virginia, in 1974. She worked as a computer programmer until 1978 when she completed an M.B.A. from the College of William and Mary. After working as a financial analyst she returned to college and completed a M.S. degree in Computer Science at William and Mary in 1989. She expects to receive her doctorate in Computer Science from the College of William and Mary in August, 1994.